A Hybrid Shared Memory/Message Passing Parallel Machine[†]

Matthew I. Frank and Mary K. Vernon

Computer Sciences Department
University of Wisconsin−Madison
Madison, WI 53706
{mfrank, vernon}@cs.wisc.edu

Abstract

Current and emerging high-performance parallel computer architectures generally implement one of two types of communication mechanisms: shared memory (SM) or message passing (MP). In this paper we propose a hybrid SM/MP architecture together with a hybrid SM/MP programming model, that we believe effectively combines the advantages of each system. The SM/MP architecture contains both a high-performance coherence protocol for shared memory, and message-passing primitives that coexist with the coherence protocol but have no coherence overhead. *The SM/MP programming model provides a framework for safely and effectively using the SM/MP communication primitives. We illustrate the use of the model and primitives to reduce communication overhead in SM systems.*

1. Introduction

Two data communication mechanisms currently dominate in existing and emerging large-scale parallel computers. In the Message Passing (or Distributed Memory) model, each processor has its own private main memory and communicates with other processors only by sending and receiving messages. The Shared Memory model comes in two main flavors. In the Static Shared Memory (or Non-Uniform Memory Access, NUMA) model, each processor is physically adjacent to a portion of global memory and communicates with other processors by reading and writing locations in its own or other portions of the shared memory. In the Dynamic Shared Memory (i.e., the Cache-Coherent or Distributed Shared Memory) model, shared data resides in a physically distributed memory and is (automatically) replicated, on demand, to provide processors with fast access to local copies.

Dynamic Shared Memory systems are easier to program in that shared read-only structures may be efficiently accessed by all processors without re-naming, and shared writable objects may migrate from processor to processor as necessary to balance the computational load. However, the data replication comes at the cost of 1) overhead for keeping the copies of shared data coherent, 2) fixed size data transfers (corresponding to a cache block or a page in virtual memory), and 3) a more complex programming model that includes some understanding of how the fixed-size copies of shared data are kept coherent. The ultimate viability of dynamic SM systems will depend on several factors, including the efficiency of the programming model, coherence mechanisms, and communication primitives, and the simplicity of the required hardware and/or compiler support. We address some of these issues in this paper.

Previous work on dynamic SM systems has focused on 1) mechanisms that guarantee that accessible local copies are coherent within some memory consistency model, and 2) the use of pre-fetching, update coherence primitives, and/or multi-threading to hide memory latencies. In this paper we take a different approach. We propose a simple message-passing facility for overlapping communication and computation in dynamic SM systems. A key idea is that a copy of a shared memory block that is not guaranteed to stay coherent can be viewed as a *message* that can be sent from one cache or memory to another. Destination processors can *read* such messages using a special read operation, so as not to accidentally read a message when reading a coherent value was intended. Like standard MP primitives, our proposed primitives and programming model expose the communication overhead of the machine so that programmers and compiler writers can easily evaluate communication costs. Unlike standard MP primitives, our proposed primitives are tightly integrated in the shared memory architecture (with only minor additional hardware support), such that shared objects need not be renamed, messages need not interrupt destination processors, and messages can be used in some cases simply to reduce coherence overhead.

In section 2 we provide some background and more detailed motivation for our proposed SM/MP primitives. In section 3 we define the new primitives and the SM/MP programming model, and in section 4 we discuss the implementation of the primitives in an SM/MP architecture. Section 5 compares our proposed primitives with various ideas in the previous literature.

[†]This work was supported in part by the National Science Foundation (CDA-9024618, CCR-9024144).

2. Background: Dynamic Shared Memory

Previous studies have observed that the dynamic shared memory programming model must include the notion of *non-uniform* memory access times, since programs developed using uniform memory access semantics have notoriously poor performance [Ande91, Cher91a, Hill90, Lin90, Mart89]. What is required is an approximate model of the coherence operations that provides a framework for assessing trade-offs between logically partitioning the data among the processes and balancing the computational load among the processes. We provide an informal model which provides a basis for our SM/MP programming model, and then comment on some disadvantages of the SM communication primitives.

Due to the benefits of read sharing as well as the overhead of updating all copies on all write operations, the following rules provide a minimal model for most high-performance coherence protocols:[1]

1. An *ordinary* read operation retrieves a *shared* copy of the page or block from main memory unless the processor already has a *shared* or *exclusive* copy.

   Side effect: If another processor has an exclusive copy of the page or block, the exclusive copy will be changed to a shared copy.

2. An *ordinary* write operation retrieves an *exclusive* copy of the page or block from main memory, unless the processor already has an exclusive copy.

   Side effect: All other *shared* or *exclusive* copies of the page or block are destroyed.

Retrieving, modifying and/or destroying copies at remote memories involves communication. As in the MP model, non-local operations are viewed as *expensive*, the exact cost being machine-specific.

Using the above model, efficient SM programs will often have similar partitioning of the computational load as their MP counterparts, due to similar models of when communication occurs. However, there are several disadvantages of the shared memory communication primitives. Ordinary read and write operations allow communication and computation to be overlapped only if multithreading is used. Prefetching may be used to address this problem, but requires the consumer(s) to request the data (at an appropriate time) and requires communication for the producer to regain write access to the data. Alternatively, an update protocol might be used to update rather than invalidate the consumers' copies. However, in hardware coherence protocols this forwarding is done each time a processor writes a word in the block; and in any case, all copies are updated whenever forwarding is done. In iterative algorithms where not all consumers need to receive the data in every iteration, the update operation has particularly high overhead. The above coherence overheads include unnecessary traffic over the interconnect as well as unnecessary memory access latencies.

[1]Note that the model can be augmented or adapted to systems with additional communication primitives such as read-exclusive and write-update operations, as well as to systems that use compiler-assisted coherence protocols in which incomplete data-dependence information may cause extra, unnecessary invalidation messages.

3. The SM/MP Programming Model

In this section we propose extensions to the dynamic SM programming model that 1) allow communication to take place with no coherence overhead and 2) facilitate maximal overlap of communication and computation. An efficient implementation of the model that can be integrated with any coherence protocol is outlined in section 4.

3.1. MP Primitives for Dynamic Shared Memory

We propose a new type of copy of a shared memory block or page, called *possibly-stale*. These copies serve as messages that can be sent from one processor to another using the MP-send and MP-read operations defined below. In some cases, the communication implemented by MP-send and MP-read will only work properly if a program can insure that sequential MP-sends to some particular destination will execute in the order they are generated. We define an MP-sync operation that can be used to guarantee this ordering. For cases where the process that computes a value does not know the identity of the consumer, and/or cases where the consumer can more effectively schedule the message transmission, we define an MP-prefetch operation that allows the consumer to fetch a message. Thus, the following primitives are defined to manipulate possibly-stale blocks:

- **MP-send**: The MP-send operation creates a new copy of a specified block or page and sends the copy to a specified processor. The new copy is of type possibly-stale. The destination processor keeps the new copy unless it already has a shared or exclusive copy of the block.
- **MP-read**: The MP-read operation returns a value from a shared, exclusive, or possibly-stale copy of the block. If the processor does not have a copy of the block, the MP-read operation retrieves a new copy of type possibly-stale.
- **MP-prefetch**: The MP-prefetch operation retrieves a copy of the block, of type possibly-stale, unless the processor already has a copy.
- **MP-sync**: The MP-sync operation completes when all previous MP-operations issued by the processor have

completed. MP operations that are issued after the MP-sync operation will be delayed until the MP-sync completes.

Typically, the processor issuing an MP-send operation has recently modified the location and thus has an *exclusive* copy of the block. As under the MP model, the processor issuing the MP-send may re-write the block without incurring any coherence overhead since the processor retains the block in state *exclusive*. The new *possibly-stale* copy is allowed to become incoherent, but can only be read using the MP-read primitive. (An ordinary read or write operation by a processor will replace a *possibly-stale* copy with a *shared* or *exclusive* copy, respectively.) From this perspective the *possibly-stale* copy is a message, and the MP-send and MP-read primitives can be used together to implement a restricted but useful form of message passing communication.

MP-send operations are treated as write operations in the memory consistency model; thus the processor may not be required to wait for these operations to complete except at specified synchronization points. Additionally, the programmer must keep in mind that the MP-read operation can return the value of a possibly stale copy, or the value of a coherent copy of the data. In this sense, the primitives must be used in a way that, for a correct SM/MP program, deleting the MP-send operations and replacing MP-read operations with ordinary reads will yield a program that is functionally the same.

3.2. Examples

We present two examples of the use of the SM/MP model. In both examples, explicit MP operations appear in the program, but might alternatively be generated by a compiler. Also, explicit MP-send operations operate on data objects; we assume that the compiler will optimize these operations by appropriately deleting duplicate operations for the same block or page. The first example is an iterative chaotic SOR algorithm where communication is reduced by calculating a schedule, either deterministically or probabilistically, which determines how often a process sends updated values to each other process (e.g. [Fuen92]). Due to the chaotic nature of the algorithm, processes are allowed to use stale values if they have not received the update. Note that MP-prefetch operations can be used instead of the MP-send operations to implement a similar communication schedule. In this example, the MP primitives are highly efficient, whereas standard prefetching or update-write operations would have significantly higher overhead.

The second example is one in which two processes communicate via a circular buffer, as is often done in pipelined and other course-grain dataflow algorithms (e.g., [Schn89]). In contrast to the previous example, the possibly-stale copies are known to be up-to-date at the time of the MP-read operations. Thus, the MP primitives are used to implement the shared buffer synchronization and to reduce coherence overheads as compared with write-update protocols and standard prefetching.

```
initialize x, A;
Spawn P threads;
task k:
  Apply-MP-read x;
  repeat until x converges
    for each i in thread k's partition of x
      x_i := f(A, x);
      compute subset-of-P to receive x_i
      MP-send (x_i , subset-of-P);
```

Figure 1. Chaotic SOR: SM/MP Algorithm (One Process)

```
Buffer[B] of items;
/* producer's buffer index */
i = 1, copy_of_i = 1;
/* consumer's buffer index */
j = 1, copy_of_j = 0;

Producer
  Apply-MP-read copy_of_j;
  repeat
    while not(i == copy_of_j)
      i := (i + 1) mod B;
      create value in Buffer[i];
      MP-send(Buffer[i], Consumer);
      every Nth iteration
        MP-sync;
        copy_of_i := i;
        MP-send(copy_of_i, Consumer);

Consumer
  Apply-MP-read copy_of_i, Buffer;
  repeat
    while not(j == copy_of_i)
      j := (j + 1) mod B;
      use item in Buffer[j];
      every Nth iteration
        MP-sync;
        copy_of_j := j;
        MP-send(copy_of_j, Producer);
```

Figure 2. Communication Using Circular Buffers: SM/MP Implementation

4. The SM/MP Architecture

The communication primitives defined in section 3 can be integrated with dynamic shared memory coherence protocols with very little increase in hardware complexity. Below we illustrate this for a generic directory-based cache coherence protocol with cache block states *exclusive*, *shared*, and *invalid*.

Table 1 specifies the actions taken by the cache in response to MP requests issued by the processor as well as MP operations that come in from the network. A new cache block state, *possibly stale* is used to implement the MP operations. No other new storage is required in the memory system. In particular, the state of the blocks in the directory is never modified by the MP operations.

MP-send requests are translated into remote MP-put requests if the processor has a shared or exclusive copy of the block. Otherwise, the cache forwards the MP-send request to the directory.

For a network MP-put request, if the block containing the word isn't present in the cache, new space is allocated for the block and the new data is stored in state *possibly stale*. If the block is in state *possibly-stale*, the new data overwrites the existing block. If the block in state *shared* or *exclusive* (or some other coherent state), the copy in the destination cache is either coherent with the copy from which the MP-put was generated, or is more recent than that copy, or is going to be invalidated by the write operation that generated the value in the MP-put. In all of these cases, it is safe to ignore the incoming MP-put.

For processor MP-read requests, the cache responds with the data if the block is in any valid state. Otherwise, the cache issues a remote MP-get request, which returns a copy of the data that will not be kept coherent by the hardware. A processor MP-prefetch request leads to a remote MP-get request unless the processor has a coherent copy of the block, in which case the MP-prefetch is ignored.

Finally, an ordinary processor read or write request purges the block if it is in state Possibly Stale, and fetches a coherent block. When a block in state *possibly stale* is deleted from the cache, no write-back occurs.

The proposed MP primitives integrate simply and easily with hardware cache coherence protocols. Implementation can similarly be integrated with distributed shared memory protocols (a.k.a. virtual shared memory) and/or with compiler-assisted coherence protocols.

Table 1: State Transitions for MP Requests in the SM/MP Coherence Protocol

Request	Cache Block State	Next State	Action
Processor MP-send	Shared or Exclusive	Unchanged	forward MP-put request to remote processor
	Possibly Stale or Invalid	Unchanged	forward MP-send request to directory
Network MP-Put	Shared or Exclusive	Unchanged	forward Ack to sender
	Possibly Stale or Invalid	Possibly Stale	insert block & forward Ack to sender
Processor MP-read	Not Invalid	Unchanged	return value
	Invalid	Possibly Stale	forward MP-get request to directory
Processor MP-prefetch	Shared or Exclusive	Unchanged	no action
	Possibly Stale or Invalid	Possibly Stale	forward MP-get request to directory
Processor Read	Possibly Stale	Shared	forward get-S request to directory
Processor Write	Possibly Stale	Exclusive	forward get-X request to directory
Network Invalidate	Possibly Stale	Possibly Stale	no action

5. Comparison with Previous Approaches

Prefetching techniques have been the principal approach advocated in the literature for reducing read latencies (i.e., for overlapping communication and computation) in large-scale SM systems. Synchronized prefetching techniques have been proposed [Good89] for simulating producer-initiated data transfers (in certain cases) in dynamic shared memory systems. These operations allow a prefetch to be issued early and to remain pending in the memory system until a new value is released by another processor. Recent examples include QOLB [Jame90], Notify [Cher91b], and cooperative prefetch [Hill92]. The advantages of the SM/MP primitives include: 1) simpler hardware support, 2) more general and efficient support for multiple consumers, 3) lower overhead and a simpler programming model in many cases (such as those illustrated in section 3.2). Whether synchronized prefetching or synchronized MP-prefetching would be desirable in an SM/MP system is an open question.

Write-update operations, which update rather than invalidate all copies of a block whenever a particular processor writes the block, are available in some systems [Leno92] and have recently been advocated as a technique for overlapping communication and computation in SM systems [Cart91, Rost93]. Our SM/MP primitives have some features in common with write-update primitives. Key differences include: 1) the explicit use of possibly stale copies allows the MP-send operations to be optimized and used more selectively than write-updates, 2) the MP-prefetch primitive, and 3) the SM/MP programming model that aids in identifying cases where the MP operations might be advantageous.

Lee and Ramachandran [Lee91] have proposed a selective WRITE-GLOBAL primitive, as well as primitives that read and write only the local cache, to support implementing a weak-consistency coherence model in software. They also propose that the software use the local-only operations for private variables, to avoid false-sharing conflicts in some cases where private and shared data are assigned to the same cache block.

Kranz et. al. have recently proposed integrating message-passing primitives into dynamic shared memory systems [Kran93]. Their principle motivations were to bundle data into large messages, and to combine synchronization with data transfer. Key differences in the SM/MP approach include: 1) the MP primitives are embedded in the shared memory hardware, and 2) SM/MP messages do not interrupt the destination processor. It remains an open question whether the SM/MP model should be augmented to include an MP-send-and-interrupt primitive.

6. Summary and Conclusions

In this paper we have proposed a set of message passing primitives for dynamic shared memory systems. We have developed the hybrid SM/MP architecture by carefully extending SM systems to incorporate key performance features of MP systems. This involved introducing a new state for cache blocks or memory pages, called possibly-stale, that can only be read by a special MP-read operation, as well as the view that blocks in this state are *messages*. The two most significant advantages of the proposed primitives are the simplicity of their implementation, and the simplicity of the model they support for overlapping communication and computation. We have also shown that the primitives allow the elimination of unnecessary coherence overhead in certain important cases. We thus believe that these primitives may be useful in improving the viability of Dynamic Shared Memory systems for parallel computing.

Key questions we are currently investigating include the performance benefits of the proposed SM/MP primitives for various applications, whether there exist examples where a variant of synchronized prefetch can improve the performance of an SM/MP architecture (thus justifying the memory system storage for pending requests), whether an interrupting MP-send operation is desirable, and whether applying existing compiler technology for distributed memory machines to SM/MP systems can yield improved performance as compared with existing compilers for dynamic shared memory systems.

References

[Ande91] Anderson, R. J. and L. Snyder, “A Comparison of Shared and Nonshared Memory Models of Parallel Computation”, *Proc. of the IEEE,* Vol. 79, No. 4 , April 1991, pp. 480-487.

[Cart91] Carter, J. B., J. K. Bennett and W. Zwaenepoel, “Implementation and Performance of Munin”, *Proc. 13th ACM Symp. on Operating System Principles*, Pacific Grove, CA , pp. 152-164, October 1991.

[Cher91a] Cheriton, D. R., H. A. Goosen and P. Machanick, “Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared Memory Multiprocessor: A First Experience”, *Int’l. Symp. on Shared Memory Multiprocessing,* Tokyo , pp. 109-118, April 1991.

[Cher91b] Cheriton, D. R., H. A. Goosen and P. D. Boyle, “Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture”, *Computer,* Vol. 24, No. 2 , February 1991, pp. 33-46.

[Fuen92] Fuentes, Y. O. and S. Kim, “Parallel Computational Microhydrodynamics: Communication Scheduling Strategies”, *AIChE Journal,* Vol. 38, No. 7 , July 1992, pp. 1059-1078.

[Good89] Goodman, J. R., M. K. Vernon and P. J. Woest, “A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor”, *Proc. 3rd Int’l. Conf. on Architectural Support for Programming Languages and Operating Systems,* Boston , pp. 64-75, April 1989.

[Hill90] Hill, M. D. and J. R. Larus, “Cache Considerations for Multiprocessor Programmers”, *Communications of the ACM,* Vol. 33, No. 8 , August 1990, pp. 97-102.

[Hill92] Hill, M. D., J. R. Larus, S. K. Reinhardt and D. A. Wood, “Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors”, *5th Int’l. Conf. on Architectural Support for Programming Languages and Systems,* Boston , pp. 262-273, October 1992.

[Jame90] James, D. V., A. T. Laundrie, S. Gjessing and G. S. Sohi, “Distributed-Directory Scheme: Scalable Coherent Interface”, *IEEE Computer,* Vol. 23, No. 6 , June 1990, pp. 74-77.

[Kran93] Kranz, D., K. Johnson, A. Agarwal, J. Kubiatowicz and B. Lim, “Integrating Message-Passing and Shared-Memory: Early Experience”, to appear *Symp. on Principles and Practice of Parallel Programming (PPoPP),* May 1993.

[Lee91] Lee, J. and U. Ramachandran, “Architectural Primitives for a Scalable Shared Memory Multiprocessor”, *3rd ACM Symp. on Parallel Algorithms and Architectures,* pp. 103-114, July 1991.

[Leno92] Lenoski, D., J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. S. Lam, “The Stanford Dash Multiprocessor”, *Computer,* Vol. 25, No. 3 , March 1992, pp. 63-79.

[Lin90] Lin, C. and L. Snyder, “A Comparison of Programming Models for Shared Memory Multiprocessors”, *Int’l. Conf. on Parallel Processing,* Vol. II , August 1990, pp. 163-170.

[Mart89] Martonosi, M. and A. Gupta, “Tradeoffs in Message-Passing and Shared-Memory Implementations of a Standard Cell Router”, *Proc. Int’l. Conf. on Parallel Processing,* Vol. III , August 1989, pp. 88-96.

[Rost93] Rosti, E., E. Smirni, T. D. Wagner, A. W. Apon and L. W. Dowdy, “The KSR1: Experimentation and Modeling of Poststore”, to appear *Proc. of the 1993 ACM Sigmetrics Conference,* May 1993.

[Schn89] Schneider, D. A. and D. J. DeWitt, “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”, *Proc. 1989 SIGMOD Conf.,* Portland, Oregon , June 1989.