Evaluating Weak Memories with Maya

Divyakant Agrawal Manhoi Choy Hong Va Leong Ambuj K. Singh
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106

Abstract

Maya is a simulation platform for evaluating the performance of parallel programs on parallel architectures with different memory coherence protocols. Rapid prototyping of different memory protocols supporting varying degrees of coherence is possible and the impact of these protocols on the performance of application programs can be studied. Implementations of existing weak memories along with some new primitives using Maya are presented. The results of running some user applications are summarized and the impact of weak memories on the efficiency of parallel programs is discussed.

Keywords: distributed shared memory, memory consistency, parallel programming, weak memories

1 Introduction

Programming parallel systems is a difficult task. The inherent concurrency of these systems poses a number of problems, namely, data placement, load balancing, race conditions, and non-determinism. Using message passing for interprocess communication adds a further degree of complexity since the programmer has to explicitly manage the shared data. Due to these considerations, the shared memory paradigm is becoming increasingly popular with hardware architects and application programmers. In order to decrease the latency of accessing shared memory, processes keep local copies of shared variables. This leads to the *cache coherence problem* of keeping multiple copies of a shared variable consistent with each other [25, 9]. The protocol that maintains the consistency of replicated copies of the shared memory among different distributed processors is called the *distributed shared memory protocol* [18] and the shared memory viewed by the programmers is called *distributed shared memory* (DSM). Several definitions have been proposed for relaxing the degree of coupling between the copies of shared variables. In general, the degree of coupling is closely related to the latency of memory access and to the cost of maintaining consistency.

Most definitions of shared memories abandon atomicity of access, the usual correctness condition in sequential systems. Examples of such non-atomic memories are *causal memory* [2] and *pipelined random access memory* [19], and memories based on the notions of *release consistency* [11], *entry consistency* [5], and *hybrid consistency* [4]. In addition, some of these definitions are specific to a particular architecture. Though these non-atomic memories improve the execution efficiency, their impact on the programming model is far from clear and there are a number of unresolved questions: Which memory is ideal for a given application? Which memory is suitable for a given architecture? Which protocol should be used to implement a given memory? How is the correctness of a program ensured in the presence of weak memories? At Santa Barbara, we are developing a system called Maya[1] to answer these questions.

Maya is a testbed that permits experimentation with different shared memories and their protocols, different user applications, as well as modeling and simulation of different architectures. Internally, it uses a portable message passing interface, which allows it to be ported quite easily to a wide variety of machines, including heterogeneous architectures. It operates principally in two different modes: the *simulation* mode and the *native* mode. The simulation mode performs a simulated execution of a parallel program on the target architecture with a specific memory coherence protocol. In the native mode, the target and the host architectures are identical. A direct execution of the protocol is carried out and no simulation is performed. In either mode, it is easy to replace one memory coherence protocol by another. Users of Maya write their programs by using an extension of C that enables processes to share variables and presents an abstraction of shared memory to the programmers.

This paper describes the results of our experiments with distributed shared memory based on three different notions of consistency: *sequential consistency* [16], *causal memory* [2, 3], and *pipelined random access memory* (PRAM) [19]. Since weak memories reduce the amount of implicit synchronization, explicit primitives are needed in order to synchronize user processes. Maya supports locks and barriers as explicit synchronization primitives. We consider two different user applications: synchronous iterative linear equation solver [2] and Cholesky factorization of a sparse matrix [21]. The first problem is solved using a coordinator and a number of worker processes. Each worker is responsible for updating a block of the input array. Proper handshaking between the coordinator and the workers ensures that the worker processes obtain the correct values for the current contents of the input array when updating their individual blocks. In the case of Cholesky factorization, the input matrix is first symbolically factorized and an elimination tree of columns is then constructed [10]. Next, the columns are partitioned between the processes and the actual factorization is carried out. We concentrate on this phase of computation during our experiments, since this is the most time consuming phase. We ran our experiments on a network of workstations and an Intel Paragon with similar results.

Maya is closely related to Munin [7, 14] at Rice and Midway [5] at CMU. Munin is one of the first DSM systems that supports multiple coherence protocols within the same system. Shared variables are classified according to their access pattern and Munin uses this information to maintain the coherence for

[1] Maya is borrowed from Sanskrit and means *illusion*.

each variable. Munin supports sequential consistency as well as eager and lazy implementations of release consistency. Midway is a DSM system that is capable of supporting multiple notions of consistency: release consistency, processor consistency, and entry consistency. Maya and Midway differ from Munin in that they are capable of supporting different notions of coherence within the same program. Maya differs from Midway in that it supports a different class of consistency definitions. Furthermore, Maya is capable of simulating different parallel architectures on a given host machine. This enables us to evaluate the performance of parallel programs and DSM protocols on different parallel architectures.

The main contributions of the paper are as follows. We show that substantial performance gains can be achieved by using weakly consistent memory for a number of applications. In the process of implementing weakly consistent memory, we devise a number of useful optimizations for reducing storage and communication overhead. We also generalize the notion of weak consistency to arbitrary objects and illustrate the usefulness of this idea. Finally, we consider the relationship between communication and synchronization and define a number of new synchronization primitives. The rest of the paper is organized as follows. Section 2 introduces different kinds of weak memories. Section 3 contains a brief overview of Maya. Sections 4 and 5 outline the implementation of different memory protocols in Maya. The performance results are detailed in Section 6 and we conclude with a brief discussion in Section 7.

2 Consistency Requirements of Distributed Shared Memories

The most widely accepted coherence requirement in the shared memory programming paradigm is the assumption of sequentially consistent memory [16]. It is the extension of the memory requirement from a single-processor environment to the multi-processor environment. It requires that the result of any execution be as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. In other words, it should be possible to construct a sequential global history by interleaving the individual process histories. A notion related to sequential consistency of memory is *linearizability* [13] or *atomicity*. Linearizability strengthens sequential consistency by requiring the preservation of operation orderings across different processes in the construction of the sequential global history. Thus, if there is only a single copy of every object and the read and write operations are implemented as remote procedure calls, then linearizability (and therefore, sequential consistency) is automatically enforced.

Pipelined random access memory (PRAM) was introduced by Lipton and Sandberg [19]. In this model, processes maintain a complete copy of the globally shared memory. A process reads and updates shared variables in its local copy. The updates are later broadcast to other processes where they occur asynchronously. The broadcasts of the updated values are assumed to occur along point-to-point FIFO channels. Note that this definition allows multiple copies of the same variable that reside at different processes to have different values. However, due to the FIFO channel assumption, all updates performed by a process are observed in the same order at any other process.

Recently, Ahamad, Burns, Hutto, and Neiger [2, 3] have proposed *causal memory* based on the idea of causality of reads and writes [15]. In this definition, events belonging to the same process are causally related by the *program order* and a read event is causally related to the write whose value it returns. A system execution is acceptable if and only if the causal order is acyclic and the values read are consistent with the causal order. Causal memory is reminiscent of causal message delivery in distributed systems just as PRAM is reminiscent of ordered message delivery [6]. For the case of two processes, causal memory becomes equivalent to PRAM.

The simple program fragment shown in Figure 1 can be used to distinguish between the above kinds of shared memory. There are three processes p, q, s and three shared variables x, y, z. The following history of the simple program is sequentially consistent:

$$\begin{array}{lll} p & :: & w(y,1)\ r(x,1) \\ q & :: & w(x,1)\ r(y,1)\ r(y,1)\ w(z,1) \\ s & :: & r(z,1)\ r(y,1) \end{array}$$

initially $x = y = z = 0$

p ::	q ::	s ::
$y := 1;$	$x := 1;$	$read(z);$
$read(x)$	$read(y);$	$read(y)$
	$read(y);$	
	$z := 1$	

Figure 1: A simple program

This is because the individual process histories can be interleaved into the following legal sequential global history.

$$w_p(y,1)\ w_q(x,1)\ r_p(x,1)\ r_q(y,1)\ r_q(y,1)\ w_q(z,1)\ r_s(z,1)\ r_s(y,1)$$

Next, consider the following history.

p :: $w(y,1)\ r(x,0)$
q :: $w(x,1)\ r(y,0)\ r(y,1)\ w(z,1)$
s :: $r(z,1)\ r(y,1)$

It is not sequentially consistent since in any interleaved global history, either the read operation by p returns an inconsistent value or the first read operation by q returns an inconsistent value. The history is however causally consistent since the resulting causal ordering, given in Figure 2, is acyclic.

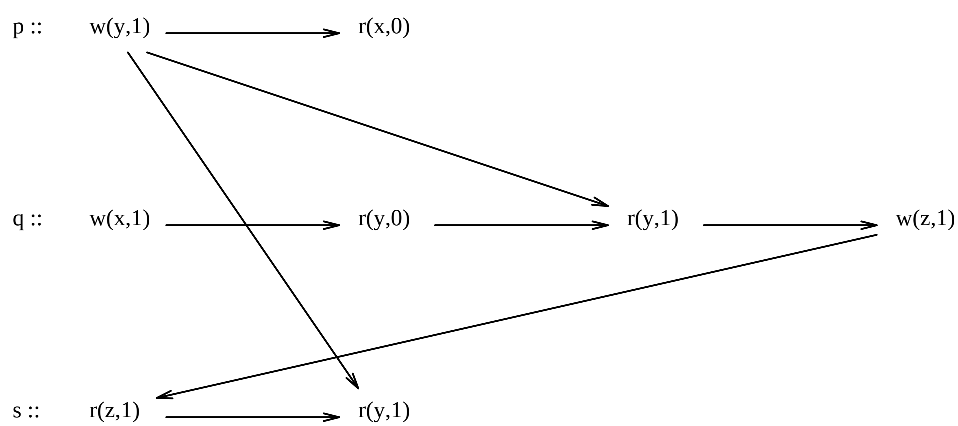


Figure 2: The causal ordering of read and write operations

Finally, consider the following history.

p :: $w(y,1)\ r(x,0)$
q :: $w(x,1)\ r(y,0)\ r(y,1)\ w(z,1)$
s :: $r(z,1)\ r(y,0)$

It is not causally consistent since the second read by s returns a value of 0 instead of the value 1 that is determined by the causal ordering

$$w_p(y,1) \rightarrow r_q(y,1) \rightarrow w_q(z,1) \rightarrow r_s(z,1) \rightarrow r_s(y,?).$$

However, the history satisfies the definition of PRAM.

We end this section by briefly discussing some other consistency conditions. *Release consistency* [11] classifies the shared accesses into *non-synchronization* and *synchronization* operations. Synchronization operations are further classified into *acquire* and *release* operations. An acquire operation essentially signals that the shared data is needed and a release operation signals that the shared data is available. Consistency is maintained only across synchronization operations. *Entry consistency* [5] strengthens release consistency by associating a unique synchronization object (e.g., a semaphore) with each critical section, i.e., a matching pair of acquire and release operations. A recent paper by Sandhu, Gamsa, and Zhou [22] advocates *shared regions*, an idea similar to entry consistency. Other definitions of memory consistency in the literature include *processor consistency* [12], *weak ordering* [8], and *hybrid consistency* [4].

3 An Overview of Maya

Maya is designed to be a versatile tool for parallel programming with a variety of features. First, it is capable of simulating a target parallel architecture on a different host machine. Second, it is useful as a programming environment for rapid prototyping of parallel programs. Finally, it is intended as a test-bed for experimenting with protocols for different kinds of shared memories. To provide these features, Maya operates either in the *simulation* mode or in the *native* mode. In the simulation mode, Maya simulates the execution of the user program on a given target architecture using a specific memory coherence protocol. In the native mode, Maya executes the user program on the host architecture again using a specified memory coherence protocol.

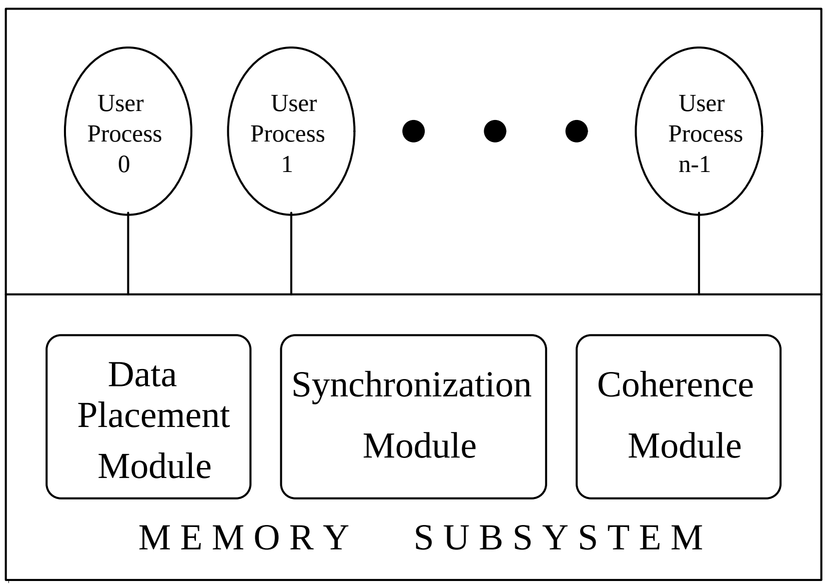


Figure 3: The logical organization of Maya

The logical organization of Maya in either mode is depicted in Figure 3. Users of Maya write their programs by using an extension of C that enables processes to share variables. Shared variables are declared explicitly and currently we restrict such sharing to the primitive types in C as well as to arrays composed of these primitive types. A pre-processor (currently under design) scans the user parallel program to identify the shared variables and arrays. These are allocated in the memory subsystem and the memory allocation is done with pages of a fixed size. The pre-processor also translates the accesses to the shared variables in the user program to standard C macro calls supported by the memory subsystem in Maya. The memory subsystem is composed of three modules: the data allocation module, the synchronization module, and the coherence module. The data allocation module is responsible for allocating space in the memory subsystem for the shared data and cooperating with other memory subsystems to partition the data. The aim is to reduce communication overhead in the sharing of the data and to minimize the effect of false sharing. At present, the data allocation algorithm is relatively straightforward. A round-robin manager-based placement strategy is adopted. On the average, each node manages $1/n^{th}$ of the total number of shared pages in an n-node system. Several user-controlled directives, which allow the declaration of possible conflicting data items involved in concurrent accesses, are provided to reduce false sharing. The pre-processor translates data placement directives into data placement calls. The synchronization module provides primitives for process synchronization: both *explicit* and *implicit*. Explicit synchronization includes primitives such as *barriers* and *locks* that can be separated from the issues of data coherence. Implicit synchronization refers to the synchronized starting of user processes and the initialization of shared variables. The coherence module enforces the required level of coherence for shared data by executing the specific memory coherence protocol when shared data is accessed. This is the main ingredient of a distributed shared memory protocol.

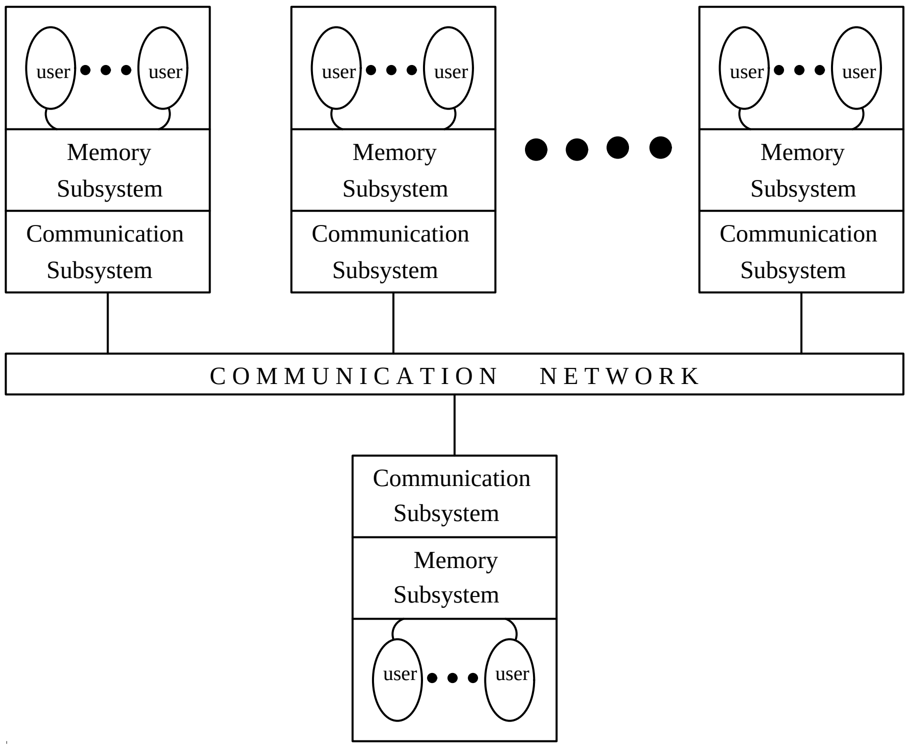


Figure 4: The physical organization of Maya

Figure 4 illustrates the physical organization of Maya. We assume that the host architecture consists of a set of processors each with its private memory (referred to as a node) connected by a network. The Maya environment on each node consists of three layers: the user program or application layer, the memory subsystem layer, and the communication subsystem layer. The memory subsystem in Maya acts as an interface between the user processes and the network to support shared memory in a distributed environment. As discussed above, the accesses to the shared variables result in macro calls to the memory subsystem. Currently the interface between the users and the memory subsystem is based on messages. The memory subsystem executes the underlying distributed shared memory protocol by communicating with the memory subsystems on other nodes. The memory subsystem in Maya is extensible. We have developed a library to support distributed shared memory protocols based on *sequentially consistent memory*, *causal memory*, *pipelined random access memory*, and *entry consistency*.

The primary responsibility of the communication subsystem is to facilitate communication among the memory subsystems, which cooperate with one another to implement distributed shared memory. The communication subsystem in Maya is based on the message passing library PVM [26]. Since PVM is available on most distributed memory architectures, Maya can be supported on a variety of hardware platforms. Another advantage of using PVM is that it can be used in a network of heterogeneous Unix compatible machines since PVM uses an external message passing standard. Although PVM imposes certain buffer management and daemon computation overhead, we consider fast prototyping and portability more important. We are investigating alternatives such as asynchronous or parallel I/O to eliminate these overheads.

Maya is designed so that it can be configured as a simulation environment for parallel architectures. Functioning as a simulator, Maya transforms the user program and the memory coherence protocol into simulation programs using a set of macros. Monitoring code is inserted for each communication event in the coherence protocol and cycle counting code is added to maintain timing information in the simulated environment. Scheduling of user processes residing at the same node is handled by the memory subsystem, which cooperates with the *network manager*. The network manager, currently implemented as a centralized process, executes a parallel discrete event simulation over the host machine nodes and network. Two major components exist in the network manager: message delay model and communication event scheduler. The message delay model is used to determine the arrival time of a message over the target architecture communication network. The communication event scheduler is used for scheduling non-local simulation events such as transmissions and deliveries of messages. Further discussion of the simulation aspects of Maya can be found in [1].

4 Implementation of Weak Memories and Synchronization Primitives

Maya manages the shared memory in terms of a set of pages that are allocated by the system during initialization. Small objects such as integers and floating point variables are placed on the same page; large objects such as arrays of integers may span several pages. Synchronization objects such as locks and barriers can also be grouped into pages. Each page can house either ordinary shared objects or synchronization objects but not both. This is achieved by using a special function that forces a page break even when the current page being allocated is not full. Each page is managed by a specific process that is determined statically. This is called the *manager* of the page. In order to achieve a good load balance, the pages are currently mapped to managers using a round robin assignment. For sequentially consistent memory protocols, this manager maintains the directory for the pages. The basic interface between the user program and the memory subsystem is via a set of macros defined for the specific coherence protocol. These macros are defined using Maya primitives *sim_request* and *sim_getreply* that send a request to the memory subsystem and get a reply from the memory subsystem respectively. Currently, these primitives are mapped into PVM message transmission routines in the native mode and into simulation routines in the simulation mode. The memory subsystem executes the following loop: wait for a message (either a request from the user program or a coherence message from other memory subsystems), update any needed local data structures, and invoke the appropriate message handler. The message handlers are specific to the coherence protocols and form the cases of a switch statement. Next, we describe the implementations of the coherence protocols and the synchronization primitives.

4.1 Sequentially Consistent Memory

The sequentially consistent memory protocol supported by the current Maya implementation is the directory-based dynamic ownership protocol using invalidations [18]. At any time, only a single process may have the write access privilege to a page; this process is called the *owner* of the page. Ownership of a page may change dynamically when the variables of the page are being accessed. The manager keeps track of the current owner of the page as well as its *copy set*, i.e., the set of nodes currently holding a copy of the page. When the node where a read or write request is originated has the required access right, the operation is performed and the result returned to the user process. Otherwise, the read or write request is forwarded to the manager. To serve a read request, the manager sends a copy of the requested page to the requesting node and registers the requester in the copy set. Meanwhile, the write access privilege of the owner node is revoked. To serve a write request, all copies of the page as indicated in the copy set are invalidated. When acknowledgements for all invalidations are received, the page is granted to the requester with the write access privilege. The requester also becomes the new owner of the page. This implementation of sequentially consistent memory also guarantees linearizability. We are currently examining update-based protocols for implementing sequential consistency as they may be more efficient for certain applications. However, we expect that most of our future experiments will focus on memory consistency conditions weaker than sequential consistency.

4.2 Causal Memory

The causal memory protocol in Maya is a fully replicated non-blocking implementation [2]. Both read and write operations are never blocked. In this protocol, each node holds a copy of the shared memory. Read operations are served locally by returning the value of the local copy of the memory word. Write operations are performed by writing the new value to the local copy and broadcasting update messages to all other nodes along with a vector timestamp that provides a causal ordering on the messages [20]. In particular, this vector timestamp counts the number of those messages on each node that are causally preceding the current message. When a message is sent, the field corresponding to the sending node in the vector timestamp is incremented by one. When a message is received, the field corresponding to the sender node in the timestamp is updated to be the maximum of the corresponding field of the current timestamp and the same field of the timestamp of the message, and the message is placed in a bag. Meanwhile, every message in the bag with a strictly smaller vector timestamp than the current timestamp is extracted into a list. This list includes all update messages that may precede the current timestamp. Finally, the list is sorted in a topological order based on the timestamps and incorporated into the local memory. Our implementation of causal memory timestamps each access instead of each variable or location. So, there is no storage overhead due to storing timestamps with variables or pages.

4.3 Pipelined Random Access Memory

The PRAM protocol in Maya is also a fully replicated non-blocking implementation [19]. Read operations are served locally and write operations are performed by updating the local copy followed by broadcasting update messages to remote nodes in a FIFO manner. When an update message is received, the effect is incorporated into the local copy without any delay. No vector timestamp or sorting is needed here, compared with the causal memory implementation. Optimization is possible by batching the updates and sending them together in larger messages. This preserves correctness as long as all messages are eventually sent.

The storage overhead of full replication in causal memory and PRAM implementations can be quite high and may lead to scalability problems. We solve this problem by introducing *freeze* operations with which full replication can be avoided. This is discussed further in Section 5.

4.4 Entry Consistent Memory

The notion of entry consistency was defined by Bershad, Zekauskas and Sawdon [5] and independently by Sandhu, Gamsa and Zhou [22]. In entry consistency, synchronization objects are used to protect a critical section of code accessing shared data. The consistency of shared data is only enforced when a critical section is entered. In the Maya implementation of entry consistency, each critical section is guarded by a *lock*. In order to leave the synchronization module intact, a separate locking mechanism is implemented as part of the entry consistency protocol. A lock is associated uniquely with each disjoint set of shared variables accessed in a critical section. These locks are mapped to and managed by lock managers that are responsible for issuing grant messages. Communication of updated values is performed in a lazy manner [5, 14]: when a lock is acquired by a process, the current values of shared variables associated with the critical section are sent to the process together with the lock grant message. The acquirer then updates the local memory and enters the critical section. When the lock is released, the lock manager is informed so that the next process may enter the critical section.

4.5 Synchronization Primitives

Weak memories derive their advantage by making all synchronization explicit, thus separating the needs of communication from that of synchronization. While weak memories supply the mechanisms for communication, synchronization primitives are needed in order to harness the efficiency of weak memories. We have implemented two such primitives: locks and barriers. A lock ensures exclusive access to a section of code. However, it does not guarantee any data coherence (as in entry consistency). A barrier is a synchronization point for a set of processes. A process participating in a barrier waits until all the participating processes reach the barrier. Again, no coherence of shared data is guaranteed at a barrier. It is up to the programmer to ensure data coherence by using appropriate kind of memory or by a judicious use of locks and barriers. Currently, locks and barriers in Maya are implemented in a centralized manner. Each lock and barrier is assigned to a manager that manages the resource. For example, all lock requests are sent to the manager of the lock, which queues up these request when the lock is in use. Similarly, the manager of a barrier counts the number of processes reaching a barrier and sends a signal to each participating process when the required number of processes have reached the barrier.

5 Performance Optimizations

In this section we discuss some schemes for improving the performance of distributed shared mmeory in Maya: the implementation of specialized synchronization primitives, the use of *freeze* operations for reducing the number of messages, the use of bulk access, and the object-orientation of shared memories. Experimental results (in Section 6) show the effectiveness of these concepts.

5.1 Specialized Synchronization Primitives

In the course of experimenting with user applications, we discovered some useful variations of the standard synchronization primitives. In particular, we introduced the concept of a barrier in the context of PRAM, called the *P-barrier*. Besides ensuring the synchronization of participating processes like a traditional barrier, a P-barrier imposes an additional requirement. It requires that all updates initiated by any participating process before reaching the barrier are observed by any other participating process before leaving the barrier. In order to implement it, every process counts the number of messages it has broadcast prior to reaching the barrier. This count is sent to the barrier manager which in turn broadcasts the vector of message counts to each participating process once all processes have reached the barrier. Each participant then waits until all messages indicated in the message count vector have been received and processed. In a similar vein, the concept of a *C-barrier* can be defined for causal memory. The concept of a *P-lock* ensures that when a process acquires the lock, it is informed of all updates performed since it last acquired the lock. This notion is closely related to release consistency [11]. However, the coherence of memory is not ensured across processes since the updates may be performed in different orders.

5.2 Reducing the Number of Messages

Since the pattern of data sharing is usually not global and limited to only a few processes, the number of messages sent in the implementations of PRAM and causal memory can be reduced greatly by sending new values only to processes that need them. This is implemented by the idea of *freeze* operations. To support freeze operation, each copy of an object has an additional field containing the list of subscribers to the object. A freeze operation is implemented by broadcasting a message to other processes to drop the requesting process from the subscription list. Later, instead of broadcasting an update message to every process, update messages are only sent to the subscribers of the object. Though this needs extra space to store the subscription list, overall memory requirements can actually be reduced since unwanted duplication of memory is avoided.

5.3 Bulk Access

The cost of accessing shared memory from a remote site can usually be reduced with bulk access, i.e. accessing data in blocks instead of individually. For example, in Li and Hudak's implementation [18], a page of data is retrieved whenever a page fault occurs. However, choosing a proper page size is difficult. Using a large page size may lead to *false sharing* and using a small page size may not be efficient. In Maya, we allow the user to specify the portion of data structure that should be treated as a single unit. Since the shared memory is divided into pages, the smallest set of pages containing the specified unit is accessed in response to a read/write request by the user. Adding bulk accesses to a DSM system does not affect the memory consistency scheme because the responsibility of using bulk accesses lies with the user; the current implementation however allows bulk accesses only for data structures allocated contiguously.

5.4 Object-oriented Memory

Object-oriented distributed shared memory is a natural generalization of distributed shared memory systems [17]. It also leads to better performance for certain applications. For example, in Cholesky factorization, a considerable amount of locking/unlocking overhead can be eliminated by treating the array elements as abstract objects providing increment/decrement operations. Herlihy and Wing have extended the idea of atomic variables to abstract objects [13]; the resulting condition is called *linearizability*. This strong consistency requirement for objects can be weakened in the spirit of weak consistency conditions for shared variables. For example, based on the degree of desired coherence, a counter object can be implemented using sequential consistency, causal memory, or PRAM.

The simplest protocol for implementing sequential consistency for objects is based on remote procedure calls. Only one copy of the memory object is maintained and all operations are performed atomically at the node owning the object. Another approach is to use replication and consider an operation that modifies the object state as a *write* operation. The previously described implementation for read/write variables can then be used. Furthermore, in situations where a certain type of operation occurs far more often than others, protocols that are based on the semantics of the operations can be used to improve efficiency [17]. Causal and PRAM implementations of shared objects are similar to the analogous implementations of read/write variables. For example, object-oriented PRAM is implemented by replicating the objects and broadcasting any updates to other nodes along FIFO channels. These updates are then applied asynchronously to the local copies of the object as in ordinary PRAM implementations.

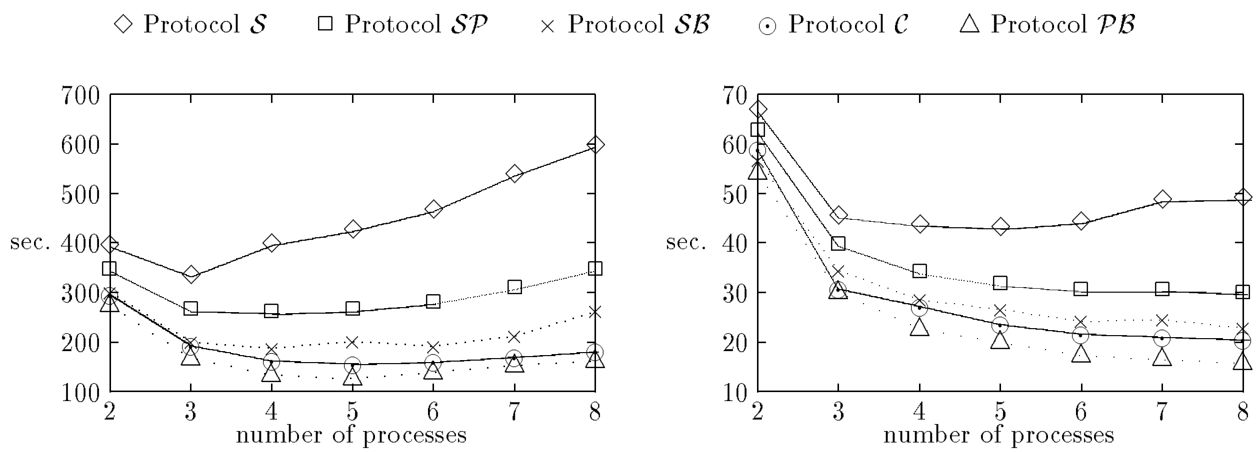
6 Performance Results

In order to compare the performance of the implementations of weak memories, we chose application programs with different patterns of data accesses. The first application is a coordinator/worker-based synchronous iterative linear equation solver [2] and the second application is a Cholesky factorization program. The first program exhibits a static data sharing and updating pattern, whereas the second program shares data statically but updates data dynamically. The first application is slightly computation bound and the second application is communication bound. We examine how these characteristics affect the performance of the programs through Maya.

6.1 Linear Equation Solver

In the synchronous iterative linear equation solver, an input array is partitioned statically into blocks, which are assigned to a number of worker processes. A worker computes the new set of values for its block by using the current contents of the array. In order to ensure a synchronous behavior, the worker processes perform a handshake with a coordinator before and after updating their block of values. The coordinator is responsible for toggling the handshaking variables and for checking the convergence of the solution. All other computations are carried out by the worker processes. In order to evaluate the relative merits of various weaker memory specifications, we develop different DSM protocols and use them in the application program. The following five strategies have been adopted.

1. Protocol $\mathcal{S}$: sequentially consistent memory without any explicit synchronization primitive.
2. Protocol $\mathcal{SP}$: sequentially consistent memory for the input and output arrays and PRAM for the handshaking variables.
3. Protocol $\mathcal{SB}$: sequentially consistent memory with barrier as a synchronization primitive.
4. Protocol $\mathcal{C}$: causal memory without any explicit synchronization primitive.
5. Protocol $\mathcal{PB}$: PRAM with P-barrier.



a) Sun Sparc/LX

b) Intel Paragon

Figure 5: Comparing weak memories using linear equation solver

We conducted the experiments on a network of SUN Sparc/LX workstations and on an Intel Paragon. The performance of the protocols is shown in Figure 5. For all five protocols, the performance improves as the number of processes is increased. However, with a large number of processes the communication network saturates, resulting in a degraded performance. Protocol $\mathcal{SP}$ performs better than Protocol $\mathcal{S}$ because no invalidation messages are sent for the handshaking variables. Protocol $\mathcal{SB}$ performs well since the handshaking variables are replaced by a barrier. The improvement of Protocol $\mathcal{C}$ is mainly because data items are always cached and invalidation messages are never sent. Protocol $\mathcal{PB}$ exhibits the best performance as it combines the merits of protocols $\mathcal{SP}$ and $\mathcal{SB}$. The relative performances of the protocols are similar on both architectures except that the U-shape of the curves is less pronounced for the Intel Paragon. This can be explained by the fact that the communication network on the Intel Paragon is much faster than the Ethernet connecting the Sun workstations.

6.2 Cholesky Factorization

Our second application, Cholesky factorization, is borrowed from the SPLASH suite [24]. It is intended to evaluate the effectiveness of object-oriented DSM. Here, the input sparse matrix is symbolically factorized and an elimination tree of columns is constructed [10, 21] in an initial phase. The time spent in the initial phase is usually negligible compared with the remaining factorization phase and is ignored here. The elimination tree captures the dependency of the columns and restricts the order of the computation: a column is updated only after all columns it depends on have been updated. For each column i, the number of columns that

i depends on and that have not been updated is kept in a shared variable $count[i]$. These variables are initialized to appropriate values during the initial phase. After column i is updated, the count variables of each of the columns that depends on column i are decremented by 1. The task of updating columns can be allocated to processes either *statically* or *dynamically*. In dynamic assignment, an idle process chooses a ready column (i.e., a column with a zero count) in a critical section. In this paper, static assignment is used for simplicity: the matrix is partitioned into fixed-size blocks, which are assigned to processes in a round robin fashion during the initial phase. Processes restrict their choice of a column only to the set of blocks assigned to them initially. Even in static assignment, the scheme for choosing the next ready column affects the performance of the algorithm significantly. We tested a number of schemes and settled on one in which preference is given to columns with smaller indices.

We considered benchmark programs from SPLASH in order to motivate the proper memory model to be used in the Cholesky factorization algorithm. This particular benchmark is designed for memory models in which locks are available and updates to shared locations are protected by locks. Consequently, if two processes try to update a column at the same time, the two updates must be sequentialized and messages must be sent between the processes. However, in the Cholesky factorization algorithm, all operations to shared variables are in the form of additions and subtractions. Since these operations commute with each other, they do not have to be sequentialized. As a result, locking and unlocking to the shared variables are not necessary. Instead, concurrent updates to shared variables could be supported. This led to the introduction of objects in memory model. The Cholesky factorization algorithm is executed with the following four protocols.

1. Protocol $\mathcal{S}$: sequentially consistent memory.
2. Protocol $\mathcal{SO}$: sequentially consistent memory with counter objects.
3. Protocol $\mathcal{PO}$: PRAM with counter objects.
4. Protocol $\mathcal{POF}$: PRAM with counter objects and freeze operations.

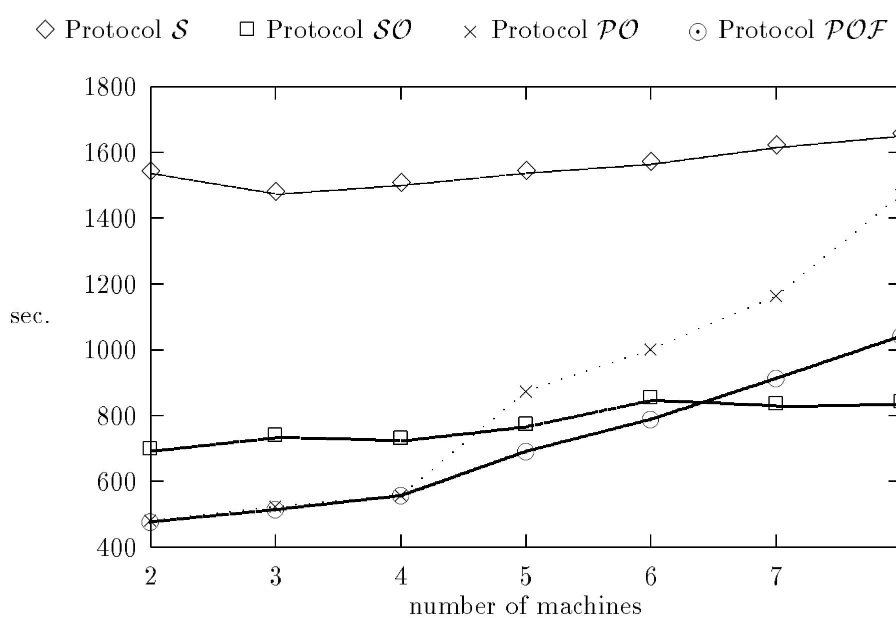


Figure 6: Comparing weak memories with objects on Sun using Cholesky factorization

Protocol $\mathcal{S}$ uses sequentially consistent memory and critical sections for updating columns. Protocol $\mathcal{SO}$ allows incrementing and decrementing a shared variable in one operation without explicitly acquiring a lock.

Also, operations are performed in a sequentially consistent manner. We expect Protocol $\mathcal{SO}$ to outperform protocol $\mathcal{S}$ since there is no overhead due to locking and unlocking. Protocol $\mathcal{PO}$ also allows incrementing and decrementing a shared variable in one operation without explicitly acquiring a lock. We expect Protocol $\mathcal{PO}$ to perform better than Protocol $\mathcal{SO}$ since PRAM memory is more efficient. Protocol $\mathcal{POF}$ is the same as Protocol $\mathcal{PO}$ except that it also allows memory locations to be frozen with respect to a process. Since Protocol $\mathcal{POF}$ is an improvement over $\mathcal{PO}$ as it uses freeze operations to reduce the number of message; we expect it to be more efficient than $\mathcal{PO}$.

We conducted our experiment on a network of Sun workstations and the results obtained are depicted in Figure 6. The performance of Protocol $\mathcal{S}$ is poor as updates to the same column are performed in critical sections and the delays due to lock acquisitions dominate the computation. We demonstrate the effectiveness of the object-oriented approach by noting an increase in speed of Protocol $\mathcal{SO}$ over that of Protocol $\mathcal{S}$ by a factor of 2 to 2.5 times. Similarly, as predicted, Protocol $\mathcal{PO}$ performs better than Protocol $\mathcal{SO}$ for small number of processes, even though the improvement is not that prominent. However, $\mathcal{SO}$ performs better than $\mathcal{PO}$ for larger number of processes. This anomaly can be explained by noting that Protocol $\mathcal{PO}$ involves the broadcasting of messages, whose count is proportional to the number of processes and starts to saturate the network when n is large. Protocol $\mathcal{SO}$ on the other hand sends only a constant number of messages for each access to a shared variable. Finally, Protocol $\mathcal{POF}$ performs consistently better than $\mathcal{PO}$ and the improvement increases as the number of processes n increases. This is because the size of the copy set is, in general, at most two in this application. As a result, the number of messages required per invalidation is reduced from $\Omega(n)$ to a constant with the use of freeze operations.

Cholesky factorization of sparse matrices is a difficult problem to solve efficiently in parallel because it is communication-bound and has an irregular communication structure. In our experiments, we used a slow communication network (Ethernet) that was subject to contention with a large number of processors. In addition, the use of the portable message passing interface PVM amplifies the communication overhead due to extra buffering and unnecessary communication between application programs and the daemon processes. This further distorts the computation/communication ratio of the application and as evident from Figure 6 impacts the scalability of the problem. A fast communication network and a small overhead communication interface seem essential for achieving any speedup for this problem. We are currently working on a faster asynchronous interface between user processes and the memory subsystem.

7 Discussion

In this paper, we have presented an overview of the architecture of Maya, concentrating on the the native mode of execution which is useful for fast prototyping of distributed shared memory protocols. The implementation of several protocols were described and experiments on two applications were conducted. The experimental results led us to several new ideas: definitions of new synchronization primitives such as locks and barriers for weak memories, object-oriented design of shared memories, and the use of freeze operations for performance optimizations.

Programming in a weak memory model may involve changing the application program so that it obtains the same result as when running on sequentially consistent memory. For the applications outlined here, these changes are minimal or non-existent. For example, for the linear equation solver, it is possible to show that causal memory leads to the same output as that obtained using sequentially consistent memory [23]. (The proof is essentially based on the commutativity of the actions of the processes.) Currently, we are investigating such well-formedness conditions for programs that use different consistency conditions such as causal memory, PRAM, and that rely on explicit synchronization primitives such as locks and barriers.

In the future, we intend to formalize the definitions of object-oriented shared memories and synchronization primitives for other weak memories. We will also support a more comprehensive library of shared memory coherence protocols. It is generally agreed that there is no single type of shared memory that is best suited for all applications. We envision a library of shared memory coherence protocols from which users can choose the best protocol for their applications. We are currently incorporating other kinds of consistency requirements in Maya in order to support such a library.

Acknowledgement

We would like to acknowledge the San Diego Supercomputing Center for providing us access to the Intel Paragon.

References

[1] D. Agrawal, M. Choy, H.V. Leong, and A.K. Singh. Maya: A simulation platform for parallel architectures and distributed shared memories. Technical Report TRCS-93-24, University of California at Santa Barbara, Department of Computer Science, 1993.

[2] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pages 9–30. LNCS, October 1991.

[3] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281. IEEE, May 1991.

[4] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 679–690, 1992.

[5] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. In *COMPCON*, 1993.

[6] K.P. Birman, T.A. Joseph, K. Kane, and F. Schmuck. ISIS - a distributed programming environment user's guide and reference manual. Technical report, Cornell University, Department of Computer Science, March 1988.

[7] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Symposium on Operating System Principles*, pages 152–164. ACM, 1991.

[8] Michael Dubois, Christoph Scheurich, and Fayé A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, May 1986.

[9] Michael Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[10] A. George and J. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26. IEEE, May 1990.

[12] J.R. Goodman. Cache consistency and sequential consistency. Technical Report CS-1006, University of Wisconsin-Madison, February 1991.

[13] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[14] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21. IEEE, 1992.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[17] H.V. Leong and D. Agrawal. Type-specific coherence protocol for distributed shared memory. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 434–441. IEEE, June 1992.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[19] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[20] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[21] Edward Eric Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization.* PhD thesis, Stanford University, Department of Computer Science, December 1992.

[22] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The shared regions approach to software cache coherence on multiprocessors. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–238. ACM, 1993.

[23] Ambuj K. Singh. A framework for programming using non-atomic variables. In *Proceedings of the 8th International Parallel Processing Symposium*, 1994, to appear.

[24] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. Technical report, Stanford, Computer Systems Laboratory, 1991.

[25] A. J. Smith. Cache memories. *Computing Surveys*, 14:473–540, 1982.

[26] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.