An Implementation of Mermera: A Shared Memory System that Mixes Coherence with Non-Coherence*

Abdelsalam Heddaya
heddaya@cs.bu.edu

Himanshu Sinha
hss@cs.bu.edu

October 16, 1992
BU-CS-92-013

Computer Science Department
111 Cummington Street
Boston, MA 02215
Phone: (617)353-8919
Fax: (617)353-6457

Abstract

Several non-coherent behaviors have been proposed to overcome the problem of poor performance of distributed shared memories. But these memories are harder to program than coherent memories. We advocate a system in which the programmer has a choice of memory behaviors giving him a trade off between performance and programming simplicity.

In our specification of Mermera, we show how one can incorporate the behavior of several types of memories, coherent and non-coherent, in one system thereby giving the programmer the choice of several behaviors. An implementation of Mermera on a network of workstations using the Isis toolkit of multicast protocols is also described. In our preliminary performance measurements we get improvements of up to a factor of 65 on certain measures for operations with non-coherent behavior compared against operations with coherent behavior. We give an example that shows how the different behaviors can be used in one program. In this algorithm we use all the behaviors provided by Mermera. We report on the performance we achieve for the application that we program using our system.

Keywords: Distributed Shared Memory, Weak consistency, Parallel Computing.

*This research was supported in part by NSF under grants IRI-8910195, IRI-9041581, CCR-8901647 and CDA-8920936.

1 Introduction

In recent years several attempts [BDG+91, LW89] have been made to use today's high performance networked workstations as a parallel computer. Most of the work in this area uses the message passing method of inter-process communication. An alternative to this is the shared memory paradigm, which has the advantage that it hides the number and identities of processors from the programmer, thereby easing the programming task.

Attempts at providing a shared memory interface [LH89, RAK88] have concentrated on keeping the memory coherent[1]. Lipton and Sandberg [LS88] have shown that it is impossible to have coherent shared memory whose access time is less than linear in the worst case delay of the communication network connecting the processors. In response, several non-coherent memories have been proposed [LS88, HA90, WW90]. In [HS92] we develop a formalism to describe all these memories.

The architecture community has also recognized the need to relax consistency constraints for shared memory multiprocessors, resulting in the proposals given in [SD88, AH90, GLL+90, Dub90]. Systems implementing some of these weaker constraints have been described in [LLG+92, BR90]. Munin [CBZ91] implements *release consistency* which guarantees sequential consistency at programmer specified synchronization points.

In [HS93] we summarize our formalism and describe the design and performance of a pilot implementation of our system, MERMERA[2], on a BBN TC2000 distributed memory parallel machine. Our approach has been to provide the programmer with a choice of a wide variety of cleanly related memory operations satisfying different ordering constraints thereby giving him an opportunity to trade off programming simplicity for performance. In [AF92], a similar approach has been taken in the specification of hybrid consistency, which defines a hybrid behavior consisting of one coherent behavior and one non-coherent behavior. In MERMERA we give a wider choice of non-coherent behaviors to the programmer.

[1] We consider coherence to mean that all processes agree on the order in which they observe memory events.

[2] *Mer* is the Latin root for memory derived from the Sanskrit root *Smar*. *Mermeros* is an ancient Greek name meaning *care laden*. Hence, the name MERMERA.

In section 2 we informally describe the different kinds of behavior supported by MERMERA and how these behaviors are combined. A formal description using partial orders can be found in [HS92]. Section 3 describes an implementation of MERMERA on a network of workstations. Section 4 summarizes the results of some preliminary performance measurements that we have done on our prototype. These measurements show that response times for non-coherent operations are independent of the number of processes sharing memory and that information flow time[3] of non-coherent operations can be 3-65 times faster than that of coherent operations. In section 5 we give examples of how MERMERA can be used to do barrier synchronization and to solve a linear system of equations. In the second example we use all the behaviors supported by our system. We discuss how our implementation can be optimized in section 6.

2 MERMERA's Memory Behavior

In this section we briefly describe each of the memory behaviors supported byMERMERA and explain how the behaviors are combined. A formal description is deferred to [Sin93]. Our system combines the behaviors of Coherent Memory [HS92], Pipelined RAM [LS88], Slow Memory [HA90] and Locally Consistent Memory [HS92]. Our model consists of processes sharing a region of their address space. These processes may be running on different processors.

We informally describe each of the memory types whose behavior is included in MERMERA. These memories are related by the hierarchy shown in figure 1 where the subset relationship implies that the set of executions allowed by a type of memory is a proper subset of executions allowed by a memory higher in the hierarchy.

Coherent Memory: All processes agree on the order of *all* writes. In other words, if a process observes[4] some writes in a certain order then no other process observes those writes in a

[3] Defined in section 4.

[4] A write, $x.w$, to location x is said to be observed by process i if the value returned by a read by process i is the value written by $x.w$ or is a value that may have been influenced by $x.w$.

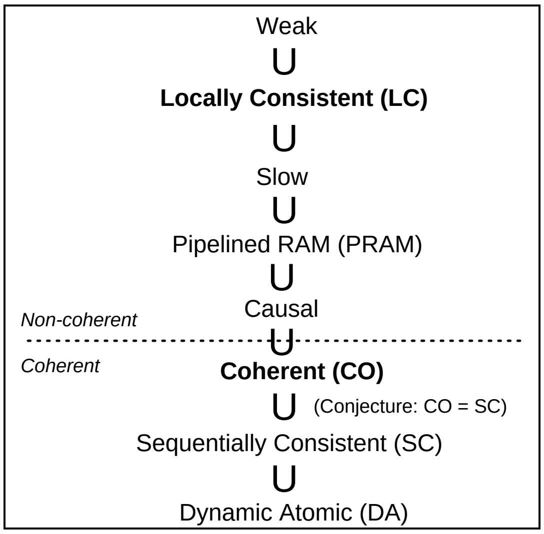

Figure 1: Hierarchy of Distributed Shared Memory.

different order[5].

Pipelined RAM (PRAM): The order of all writes by the *same* process is respected by all processes, *i.e.*, if a process performs two writes, w_1 followed by w_2, then no process can read them in the reverse order. Writes by different processes may be interleaved in different orders by different processes.

Slow Memory: All writes by *the same process to the same location* are ordered by all processes in the order they were written. Writes to different locations by the same process may be ordered differently by different processes.

Locally Consistent Memory: It appears to *each* process that all the events *it* observes are executed on a single processor in an order consistent with its program. This differs from sequentially consistent memory [Lam79] in which *all* events appear to be executed on a single processor in an order consistent with *every* process' program.

In [HS93] we argue that a system that provides a combination of these behaviors is useful. In MERMERA we provide all these behaviors and the programmer can use any of the behaviors by invoking the appropriate operation. We provide four kinds of write operations to shared memory:

[5]We conjecture in [HS92] that coherent memory is equivalent to sequentially consistent memory [Lam79].

CO_Write, PRAM_Write, Slow_Write and *Local_Write*. Each of these operations takes a *location* and a *value* as arguments. Values are read from shared memory using the *Read* operation.

If a programmer uses write operations of only one kind then the programmer can assume that MERMERA provides him with only that type of memory. If a programmer chooses to use more than one of the different kinds of write operations then the behavior is as follows. There is a global total order in which *CO_Writes* are observed by the different processes and this order is consistent with each process' program. All *PRAM_Writes* and *CO_Writes* by the same process are ordered by all processes in the order they were specified in the writing process' program. All *Slow_Writes*, *PRAM_Writes* and *CO_Writes* satisfy the correctness condition of Slow memory described above. Similarly, all *Local_Writes*, *Slow_Writes*, *PRAM_Writes* and *CO_Writes* satisfy Local Consistency.

We are considering putting additional constraints on executions that mix different types of operations. One may want a *CO_Write* to *cleave* the computation by forcing all processes to agree on whether weaker writes appear to precede or to follow the *CO_Write*. *PRAM_Write* events can have a similar effect, but only on the subset of the computation that contains PRAM or weaker writes by the same processor.

We note that the above specification of MERMERA does not impose any liveness conditions on an implementation, *i.e.*, it does not require that all writes be propagated to all processes. We plan to exploit this leeway in our implementation to enhance performance.

3 Implementation

In this section we describe an implementation of MERMERA on a network of SPARCStations running Unix. Our algorithms are update based, *i.e.*, each process has a copy of the shared memory and this copy is updated as writes occur[6]. A *Read* returns the value in the local copy. A write operation to shared memory updates the local copy and propagates the value to other processes. How this propagation is done depends on the type of the write operation.

[6]The implication of this full replication on the performance comparisons is discussed in section 4.

The specification of MERMERA does not require that all writes be propagated to other processes. Only *CO_Writes* and *PRAM_Writes* are guaranteed to be propagated to all processes. *Slow_Writes* can be propagated on a "best effort" basis, *i.e.*, the system tries to propagate them but no guarantees are given[7]. The only guarantee is that the correctness conditions will not be violated. *Local_Writes* are done only to the local copy and no attempt is made to propagate them. The weaker ordering constraints on non-coherent operations also allow us to buffer multiple writes and propagate them together.

The writes are propagated by broadcasting the values to other processes. We use the Isis toolkit [BSS91, BJ87] for our implementation because it gives us a suite of broadcasts to groups of processes which satisfy different ordering properties. The broadcasts of interest to us are *abcast*(), *fbcast*() and *mbcast*(). These primitives have different constraints on the order in which the messages are delivered to their destinations.

All messages sent using *abcast*() are delivered in the same order at all destinations, *i.e.*, there exists a total order on the order in which these messages are received by a process and this order is the same for all processes. This is exactly the property we want for *CO_Write*.

Messages sent using *fbcast*() obey a weaker constraint. Messages sent the *same* process are received by all processes in the order they were sent. However, *fbcasts* sent by different processes may be interleaved in different orders at different processes. This suffices for *PRAM_Writes*, so we use *fbcasts* to propagate them.

Messages sent using *mbcast*() do not provide any ordering guarantees[8]. We use this primitive for propagating *Slow_Writes*.

No ordering guarantees are provided among messages sent using different primitives, *e.g.*, if two messages are sent one after the other using *abcast*() and *fbcast*(), respectively, they are not guaranteed to be received in the order they were sent. We take this into consideration in our

[7]This interpretation of *Slow_Writes* has not been implemented. The current implementation guarantees propagation of *Slow_Writes*.

[8]The current implementation of Isis does deliver them in the order they were sent by the sender.

implementation and ensure that the correctness conditions above are satisfied. How we do this is explained later in this section.

Another feature of Isis that we use is its lightweight task system. This allows us to have several concurrent tasks in a user process. These tasks are non-preemptive which makes synchronizing accesses to different data structures very easy, *i.e.*, we do not have to worry about enforcing mutual exclusion on accesses to data structures. A task controls when it gives up the CPU to other tasks in the process. The delivery of a message to a process causes a task to be created. Tasks are scheduled in FIFO order. Each message names an entry point that specifies the task to invoke to handle the message.

We now explain how we combine the different operations[9]. These algorithms are given in figures 3, 4 and 5. All writes other than *CO_Writes* may be buffered. The structure of the buffer is shown in figure 2. Its size can be dynamically adjusted[10]. The $(Location, Value)$ pair of a *CO_Write* is appended to the buffer and the entire buffer is immediately broadcast using the abcast protocol to all processes (including the writer). The task that issues the operation continues only after the message has been processed by its process. This ensures that the message is processed in the global order of abcasts. In case of *PRAM_Writes* and *Slow_Writes* the local copy is immediately updated and the updates are appended to the buffer. This buffer is broadcast when it gets full or when a *CO_Write* needs to be broadcast. Furthermore, it is also broadcast when a preset timeout occurs. If the buffer is not full then the task issuing the *PRAM_Write* or *Slow_Write* can continue after the local copy has been updated but before the buffer is sent. The broadcast protocol used depends on the type of the strongest write in the buffer (according to the hierarchy in figure 1 with *CO_Write* being the strongest). The *fbcasts* and *mbcasts* are sent asynchronously to all processes except the writer. The abcasts and fbcasts are reliable broadcasts in the sense that the eventual delivery of messages sent using these broadcasts is guaranteed. We use mbcasts, which happen to be reliable, for *Slow_Writes* but a "best effort" policy does not require this reliability. As discussed in section 6

[9] In section 6 we mention some optimizations that can be made to our implementation.

[10] For the sake of simplicity we allow each write to write only floating point numbers to shared memory. This can be modified to allow writes that modify an arbitrary number of bytes in shared memory.

SenderId	
SeqNo	Sequence #, Consistent with the order in which buffers are sent by *this* process.
LastRelSeqNo	SeqNo of last reliable (*abcast* or *fbcast*)broadcast.
Type	Type of strongest write in this buffer.
Location	**Value**
...	...
...	...

Figure 2: Structure of buffer. This buffer is the message that is broadcast.

we may switch to an unreliable broadcast protocol for propagating *Slow_Writes*. *Local_Writes* are applied only to the local copy, although their specification permits their propagation.

When an update message is delivered to a process a task specified by the *update_memory* function (figure 5) is created. This task is responsible for applying the updates in the message. If applying the update will violate any of the correctness conditions then that buffer is put in a waiting list. If the buffer is a CO buffer then it is also enqueued in a list of CO buffers. Whenever an update is applied the waiting list is checked to see if updates from any other buffer can be applied without violating the correctness conditions.

Roughly speaking, updates from a buffer are applied if all reliable messages[11] from the sender of the buffer have been received (and their updates applied). In case of CO buffers an additional check has to be made to ensure that updates from all previous (in the total order of abcast messages) abcast messages have been applied. Slow buffers are forced to wait only if the receiver is waiting for a reliable message from the same process. If a Slow buffer arrives after updates from a subsequent (in the order of buffers sent by its sender) buffer have been applied then its updates are ignored.

[11]That is, messages sent using *abcast* or *fbcast*.

```
CO_Write(int Loc, int Val)
{   Append (Loc, Val) to Buffer;
    Buffer.Type = CO;
    broadcast(Buffer);
    Wait for update-received signal.
}

PRAM_Write(int Loc, int Val)
{   Update local copy of Loc.
    Append (Loc, Val) to Buffer;
    if (Buffer.Type == Slow)
        Buffer.Type = PRAM;
    if (Buffer is full)
        broadcast(Buffer);
}

Slow_Write(int Loc, int Val)
{   Update local copy of Loc.
    if (Buffer is empty) BufferType = Slow;
    Append (Loc, Val) to Buffer;
    if (Buffer is full)
        broadcast(Buffer);
}

Local_Write(int Loc, int Val)
{   Update local copy of Loc;}
```

Figure 3: Pseudo C code for write operations

```
broadcast(Buffer)
{   static int ReliableSeqNo = 0;          /* Sequence # of the last reliable broadcast*/.
    static int SequenceNo = 0;
    Buffer.SeqNo = SequenceNo;
    Buffer.LastRelSeqNo = ReliableSeqNo;
    switch (Buffer.Type)
    {   case Slow:
            Asynchronous-mbcast(Buffer);                        /* To all but self */
            break;
        case PRAM:
            Asynchronous-fbcast(Buffer);                        /* To all but self */
            ReliableSeqNo++;
            break;
        case CO:
            Asynchronous-abcast(Buffer);                  /* To all including self */
            ReliableSeqNo++;
            break;
    }
    SequenceNo++;
}
```

Figure 4: Pseudo C code for *broadcast*().

4 Preliminary Performance Results

In this section we report on the results of some preliminary performance measurements of our implementation of MERMERA on an Ethernet network of six dedicated SPARCstation 1+s running SunOS Unix. Each of the processes in our experiments runs on a different workstation. We measure two quantities for each process doing 100 write operations to the shared memory. All processes do their writes at approximately the same time between two barrier synchronization calls described in section 5. First, we measure the response time at each process for these 100 writes. Second, we measure the time it takes for all processes to have received all 100 writes from all participating processes. We call this the *information flow time*. To help us perform this measurement we do a barrier synchronization after a run of 100 writes. The parameters that are varied are: the number of processes sharing memory (from 1 to 6) and the size of the buffer that is used to buffer the non-coherent writes (1, 10, 100, 1000 location-value pairs). Tables 1 and 2 list our measurements so far.

```
                      /* WaitList[i] contains the RelSeqNo and the SeqNo of the last buffer pro-
WaitList[];           cessed from process i and a PendingBufList of buffers that have been received
                      by this process but whose updates have not been applied at the local copy */
AbcastQ;                                        /*A queue of unprocessed Abcast buffers */
update_memory(CurrentBuffer)                /* Invoked when a update message is received */
{   if ((WaitList[CurrentBuffer.Sender].LastRelSeqNo < CurrentBuffer.RelSeqNo) ||
        ((CurrentBuffer.Type == CO) && (AbcastQ.Head != NULL)))
        {   insert_in_WaitList(CurrentBuffer);          /* Inserts CO Buffers in AbcastQ also */
            return;}
    while (CurrentBuffer != NULL)
    {   Sender = CurrentBuffer.Sender;
        switch(CurrentBuffer.Type)
        {   case CO:
                WaitList[Sender].LastRelSeqNo = CurrentBuffer.SeqNo;
                if (CurrentBuffer.SenderId == MyId)
                {   Apply only the CO_Write which is the last update in the buffer.
                    Send update_received signal so that waiting task can continue. }
                else Apply all updates to the local copy
                break;
            case PRAM:
                Apply updates to local copy;
                WaitList[Sender].LastRelSeqNo = CurrentBuffer.SeqNo;
                break;
            case Slow:
                if (CurrentBuffer.SeqNo > WaitList[Sender].LastSeqNo)
                    Apply all updates to the local copy;
                break; }
        WaitList[Sender].LastSeqNo = CurrentBuffer.SeqNo;

        /* Check if any of the pending buffers can be processed */
        CurrentBuffer = NULL;
        PendingBuf = WaitList[Sender].PendingBufsList.Head
        if (WaitList[Sender].LastRelSeqNo == PendingBuf.RelSeqNo)
        {   if (PendingBuf.Type != CO)
            {   Remove PendingBuf from PendingBufList
                CurrentBuffer = PendingBuf;}
            else if (AbcastQ.Head == PendingBuf) /* CO_Buffer should be at the head of the Ab-
                                                  castQ to be processed */
            {   Remove PendingBuf from PendingBufList and AbcastQ.
                CurrentBuffer = PendingBuf;}}
        if (CurrentBuffer == NULL)
        {   PendingBuf = AbcastQ.Head
            if (WaitList[PendingBuf.Sender].PendingBufList == PendingBuf)
            {   CurrentBuffer = PendingBuf;
                Remove PendingBuf from PendingBufList and AbcastQ.}
}   }   }
```

Figure 5: Pseudo C Code for the handler for the *update_memory* message

Procs	CO_Write	PRAM_Write				Slow_Write			
		Buffer Sizes				Buffer Sizes			
		1	10	100	1000	1	10	100	1000
1	136	47	5.8	1.2	.65	44	5.5	1.2	.6
2	1000	100	18	2.7	.65	100	16	2.5	.6
3	1420	100	30	4.0	.65	86	27	3.5	.6
4	2950	100	32	5.7	.65	88	31	4.6	.6
5	4800	100	32	6.8	.65	95	30	6.0	.6
6	5800	110	33	7.5	.65	96	30	6.3	.6

Table 1: Response time in milliseconds for 100 writes. Buffer Size is measured in terms of the number of write operations that can be buffered.

Procs	CO_Write	PRAM_Write			Slow_Write		
		Buffer Sizes			Buffer Sizes		
		1	10	100	1	10	100
1	140	50	9	4.1	47	8.8	4.1
2	1000	260	50	30	260	47	26
3	1600	500	100	50	360	84	36
4	3040	700	140	80	600	130	50
5	4950	1000	200	100	900	180	80
6	6500	1300	240	120	1200	230	100

Table 2: Information flow time in milliseconds for 100 writes.

The buffer size has no significant effect on the performance of *CO_Writes* because no buffering is done for them. In case of *PRAM_Write* and *Slow_Write* the buffer size is the most important factor that affects the response time and the information flow time. This is because the buffer size determines the number of messages that are sent and received. The response time for the non-coherent writes is almost independent of the number of processes because the broadcast of their writes is done asynchronously. We do not yet know why this is not true for a buffer size of 100. When the buffer fills up it is scheduled for broadcast and the writing task can continue. On the other hand, a task doing a *CO_Write* has to wait till the message is delivered to its process before it can continue.

The information flow time includes the time spent in doing the asynchronous broadcasts and the time spent in executing tasks that are spawned as a result of incoming updates. The number of messages sent and received is a significant determining factor for information flow time. This number depends on the buffer size. However, if we are doing 100 writes then all their updates can be sent in a buffer of size 100. Therefore, increasing the buffer size beyond 100 does not have any effect on information flow time. The response time and information flow times for *CO_Writes* are close because the FIFO scheduling of tasks by Isis' light weight task system. If a task is waiting for its process to receive its broadcast then all tasks resulting from update messages that arrive in the meantime will be executed before the waiting task continues. As a result of this, by the time a process returns from it 100 writes it has already processed many of the writes by other processes.

Table 2 shows that for a fixed buffer size the information flow time of non-coherent memories depends linearly on the number of processes sharing memory. On the other hand the information flow time of coherent writes grows faster than linearly as the number of processes is increased. Information flow times for *CO_Writes* is 3-50 times slower than *PRAM_Writes* (depending on the buffer size used) and 3-65 times slower than *Slow_Writes*. The factor increases as the number of processes increases.

One may argue that this comparison of the information flow time of coherent writes and non-coherent writes is unfair because few coherent memories are implemented using full replication. More efficient implementations such as directory based schemes (see [CKA91] for an example) exist. Our response to this argument is that there are some applications (one of which is presented in section 5) which intrinsically generate a message traffic that will be similar to the traffic generated by using full replication to implement shared memory. The characteristic of such applications is that all participating processes read the values written by all other processes regularly. Moreover, one may be able to devise more efficient implementations of non-coherent memory by exploiting techniques developed for cache consistency.

The slightly better performance of *Slow_Writes* with respect to *PRAM_Writes* can be attributed to the lower cost of *mbcasts* compared to that of *fbcasts*. We expect this difference to increase with the optimizations suggested in section 6.

```
barrier()
{   while ((i < NumOfProcs) && (Read(i + 1) == false))
         isis_accept_events();
    CO_Write(i, true);
    while ((i > 1) && (Read(i) == true))
         isis_accept_events();
    CO_Write(i + 1, false);
}
```

Figure 6: Pseudo C code for process i to implement barrier synchronization. Locations 1 thru (NumOfProcs + 1) in shared memory are reserved for use in barrier synchronization. Calls to isis_accept_events allow updates made by other processes to be processed.

5 Using MERMERA

In this section we show how the operations of MERMERA can be used to achieve barrier synchronization. We then use all the behaviors of MERMERA to solve a system of linear equations using an asynchronous iterative method [BT89][12].

5.1 Barrier Synchronization

Let there be n cooperating processes running concurrently. Each of them makes k calls to the function $barrier()$. The requirement is that their i^{th} call to $barrier()$ return only after all of the n processes have made the i^{th} call to $barrier()$. Our algorithm (figure 6) uses $n+1$ shared flags, all initialized to *false*. The barrier is cleared after two phases in which those flags are all set, and then reset. In the first phase, every process sets its own flag, using a *CO_Write*, after waiting for all processes with a higher ID to set their respective flags. Thus the flags are set in decreasing order of process ID's. Dually, in the second phase, the flags are reset in *increasing* order of process ID's. This guarantees that no process will clear the barrier until all others have reached it, and that no process will break the barrier synchronization if it invokes *barrier*() while some processes are still in it.

[12] The potential performance advantage of asynchronous iterative methods was first established by Baudet in [Bau78].

We cannot use *Slow_Write* for to modify the flags because they are not guaranteed to be applied at other processes. Loss of a write in this case can result in a process waiting forever in one of the two **while** loops in the algorithm. We do not use *PRAM_Write* because it may get buffered and propagated after some delay. In the meantime the task doing the barrier cannot do anything. A process has nothing to gain by buffering the first of the two writes in the algorithm. The second write can be a *PRAM_Write* and it will allow process i to propagate the write later but this will be at the expense of process $(i+1)$ having to wait longer to cross the barrier.

5.2 Solving a System of Equations

In this section we outline a method to solve a linear system of equations in m variables using n processes running in parallel and sharing memory using MERMERA. We use the totally asynchronous algorithms described in [BT89] to solve a system $Ax + B = 0$. A is an $m \times m$ matrix and B is a vector of size m. We want to find the fixed point of the following iteration.

$$x_i = -(b_i + \sum_{j=1}^{i-1} a_{ij} \times x_j + \sum_{j=i+1}^{m} a_{ij} \times x_j)/a_{ii}$$

In [HS92] we proved that Slow Memory is sufficient for the convergence of this asynchronous iteration provided A and B satisfy certain numerical conditions. But we need stronger behavior of the memory for *detecting* the convergence of the iteration. Our algorithm is shown in figure 7.

We partition x among the n processes so that each process does the above iteration for x_i in its partition. x is in shared memory while A and B are in each process' local memory. Upon computing a new value for x_i a process writes it to shared memory using *Slow_Write* which guarantees that the value will be written to the local copy but it may not be propagated to all other processes. It may be propagated to some or none of them. A call to *isis_accept_events()* gives a chance for updates from other processes to be applied to the local copy of shared memory and also for events like timeouts to be handled. The call to *barrier* ensures that all processes have the values of x that satisfied the local termination at the processes that computed them. The *global_termination()* check is done by doing another iteration for all of x and the new values are written to the local copy using *Local_Write*. The function returns *true* if the sum of absolute difference of the new value

```
Epsilon = 0.0001                                              /* Accuracy desired */
do
{ do
  { AbsoluteDiff = 0;
    for (i = MyLow; i < MyHigh; i++)
    { NewXi = −(b_i + Σ_{j=1}^{i−1} a_ij × x_j + Σ_{j=i+1}^{m} a_ij × x_j)/a_ii;   /* Use Read to read x_j */
      AbsoluteDiff = AbsoluteDiff + abs(NewXi - Read(XStartLoc + i));
      Slow_Write(XStartLoc + i, NewXi);
      isis_accept_events();
    }
  }while (AbsoluteDiff < Epsilon)                             /* Local termination check */
  for (i = MyLow; i < MyHigh; i++)
    PRAM_Write(XStartLoc + i, Read(XStartLoc + i));           /* Ensure that values propa-
                                                                 gate to all processes */
  barrier();                          /* Wait for all processes to satisfy local termination */
}while (!global_termination());
```

Figure 7: Program to solve a system of equations. *MyHigh* and *MyLow* are local variables that specify the part of x that this process is computing. *XStartLoc* is the index in shared memory where the first component of x is stored.

from the old value is within the desired limits of accuracy. Note that the global termination check is done by all processes.

In this algorithm we use all the behaviors provided by MERMERA. We have programmed this algorithm on our implementation (figure 7). Our preliminary measurements show that when using 5 workstations to solve a system of equations in 1000 variables it performs up to 4 times faster than a program that uses *CO_Writes* only. The solution computed for our sample (randomly generated) problem are extremely accurate. Detailed measurements are under way, and will be reported in [Sin93].

6 Discussion

We presented an implementation of MERMERA in section 3. The approach was to use broadcast primitives that approximately satisfy the conditions required by the different kinds of write operations. The messages received through the different primitives are processed by the receivers in an order that does not violate the correctness conditions in the specification. Since the receivers check that messages are processed in a certain order it is possible to use *mbcast* (or any protocol that guarantees message delivery) for propagating *PRAM_Writes*. The sequence number information in the messages is sufficient to ensure the PRAM ordering. This optimization should improve the performance of *PRAM_Writes* closer to that of our current implementation of *Slow_Writes*, which uses reliable message delivery.

We feel that the performance of *Slow_Writes* can be significantly improved by using a "best effort" policy for their propagation. Thus, one can use a broadcast protocol that tries to send the messages to the participating processes but does not verify that the messages are delivered. These messages may be received only by some or none of the processes. Such a protocol should perform significantly better than the current implementation because besides guaranteeing delivery, the current Isis implementation of *mbcast* uses a transport protocol that guarantees that messages are received in the order sent.

In our performance measurements we observed that non-coherent writes have an information flow time that is up to 65 times faster than coherent writes. We are actively seeking an answer to the question of how this affects the performance of applications that switch from coherent writes to non-coherent writes. We have implemented the algorithm described in section 5 and we are in the process of measuring and analyzing its performance. Very preliminary measurements show an improvement in performance by a factor of 4 for a program using all the behaviors of MERMERA over a program using CO_Writes alone. Both programs use 5 workstations to solve a system of equations in 1000 variables.

Our implementation uses full replication. While it will perform well for applications in which every process reads the values written by every other process it may not do well in applications that exhibit locality in inter process communication. We are interested in implementations that can take advantage of locality of communication. We are studying the possibility of an implementation that uses partial replication or invalidation mechanisms.

References

[AF92] H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors. Technical Report #719, Technion – Israel Institute of Technology, Department of Computer Science, March 1992.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 28-31 1990.

[Bau78] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *JACM*, 25(2):226–244, April 1978.

[BDG+91] Adam Beguelin, Jack Dongarra, Al Geist, Robert Mancheck, and Vaidy Sunderam. A user's guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[BJ87] K. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. 11th ACM Symp. on Operating System Principles*, Austin, Texas, Nov. 1987. Also available as technical report TR 87-811 from the Dept. of Computer Science, Cornell Univ., Feb. 1987.

[BR90] Roberto Bisiani and Mosur Ravishankar. Programming the PLUS distributed-memory system. In *Proceedings of the Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press, Los Alamitos, California, April 8-12 1990.

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, Aug. 1991.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[CBZ91] John Carter, John Bennet, and Willy Zwaenopoel. Implementation and performance of Munin. In *Proc. 13th ACM Symp. on Operating System Principles,*, October 1991.

[CKA91] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS directories: a scalable cache coherence scheme. In *Proc. 4th ACM Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California*, pages 224–234, Apr. 1991. Describes the Alewife machine.

[Dub90] Michel Dubois. Delayed consistency protocols. Technical Report CENG 90-21, Electrical Engineering – Systems Department, University of Southern California, July 1990.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 28-31 1990.

[HA90] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. 10th IEEE Intl. Conference on Distributed Computing Systems, Paris, France*, June 1990. Also available as GATech technical report GIT-ICS-89/39.

[HS92] Abdelsalam Heddaya and Himanshu S. Sinha. Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing. Technical Report BU-CS-92-004, Computer Science Department, Boston University, May 1992.

[HS93] Abdelsalam Heddaya and Himanshu S. Sinha. An Overview of MERMERA:A System and Formalism for Distributed Parallel Memory. In *Proc. 26th Hawaiian International Conference on System Sciences (HICSS-26)*, Maui, Hawaii, January 1993. Also available as Boston University technical report BU-CS-92-009.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9), September 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[LLG+92] Daniel Lenoski, James Lauson, Kourosh Gharachorloo, Wolf-Deitrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LS88] R.J. Lipton and J.S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., Dept. of Computer Science, Sep. 1988.

[LW89] Jerrold S. Leichter and Robert A. Whiteside. Implementing Linda for Distributed and Parallel Processing. Technical Report YALEU/DCSTR-715, Department of Computer Science, Yale University, April 1989.

[RAK88] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in mantaining coherence of distributed shared memory. Technical Report GIT-ICS-88/23, School of Information and Computer Science, Georgia Institute of Technology, June 1988.

[SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 15-19 1988.

[Sin93] Himanshu Sinha. MERMERA*: A Formal Framework and Implementation for Parallel Computation Using Non-Coherent Distributed Shared Memory.* PhD thesis, Boston University, Computer Science Department, 111 Cummington Street, Boston, MA 02215, In preparation 1993.

[WW90] W. E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent b-trees. Personal communication, May 1990.