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

BU-CS-93-006
(Supersedes 92-013)
June 4, 1993

Boston University
Computer Science Department
Boston, MA 02215
Phone: 617 353-8919
Fax: 617 353-6457

Abstract

Coherent shared memory is a convenient, but inefficient, method of inter-process communication for parallel programs. By contrast, message passing can be less convenient, but more efficient. To get the benefits of both models, several non-coherent memory behaviors have recently been proposed in the literature.

We present an implementation of Mermera, a shared memory system that supports both coherent and non-coherent behaviors in a manner that enables programmers to mix multiple behaviors in the same program [HS93]. A programmer can debug a Mermera program using coherent memory, and then improve its performance by selectively reducing the level of coherence in the parts that are critical to performance.

Mermera permits a trade-off of coherence for performance. We analyze this trade-off through measurements of our implementation, and by an example that illustrates the style of programming needed to exploit non-coherence. We find that, even on a small network of workstations, the performance advantage of non-coherence is compelling. Raw non-coherent memory operations perform 20-40 times better than non-coherent memory operations. An example aplication program is shown to run 5-11 times faster when permitted to exploit non-coherence. We conclude by commenting on our use of the Isis Toolkit of multicast protocols in implementing Mermera.

Keywords: Distributed Shared Memory, Weak consistency, Parallel Computing, Asynchronous Iterative Methods, Isis.

*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 harness today's high performance networked workstations for parallel computing. 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] observed 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 [AH90, GLL+90, Dub90, SD88]. Systems implementing some of these weaker constraints include PLUS [BR90]. The DASH multiprocessor [LLJ+93] and the Munin software shared memory system [CBZ91] allow sequential consistency to be broken, but only during the execution of code blocks explicitly marked by the programmer.

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 memory 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, roughly, 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*. So far as we know, Mermera is a new word.

In section 2 we informally describe the different behaviors supported by Mermera and how these behaviors are combined. A complete formal description using partial orders can be found in [HS92]. Section 3 presents an implementation of Mermera on a network of workstations. Section 4 gives the raw performance of our prototype. It shows that access time for non-coherent operations is independent of the number of processes sharing memory and that completion times[3] of non-coherent operations are approximately 20-40 times faster than those of coherent operations. Section 5 contains an example program to solve a linear system of equations exercising all of Mermera's features. The same section includes extensive measurements that show that the solver can achieve a 5-11 fold improvement in run time by using non-coherence. We discuss how our implementation can be optimized in section 7.

2 Mermera's Memory Behavior

In this section we briefly describe each of the memory behaviors supported by Mermera and explain how the behaviors are combined. A complete formal description can be found in [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.

The memories supported by Mermera 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 an order of *all* writes that is consistent with their individual program orderings. In other words, if a process observes[4] some writes in a certain order then no other process observes those writes in a different order[5].

Pipelined RAM (PRAM): The order of all writes by the *same* process is respected by all pro-

[3] Defined in section 4.

[4] Informally, 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 $x.w$.

[5] Sinha [Sin93] shows that coherent memory is equivalent to Lamport's sequentially consistent memory [Lam79].

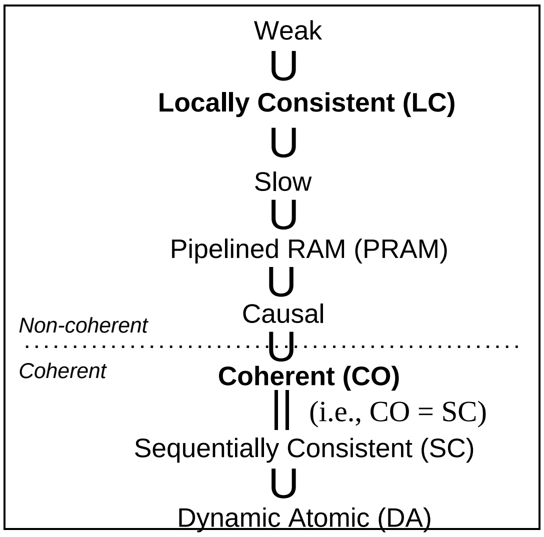

Figure 1: Hierarchy of Distributed Shared Memory.

cesses, *i.e.*, if a process performs 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 is much weaker than sequentially consistent memory [Lam79] in which *all* events appear to be executed on a single processor in an order consistent with *every* process' program.

Several classes of parallel programs have been shown to run correctly on different non-coherent memory behaviors, potentially benefitting from the superior performance of these memories. Lipton and Sandberg show in [LS88] that PRAM can be used to solve a large number of applications like FFT, matrix-vector product, matrix-matrix product, dynamic programming and other computations that are in the large class of *oblivious computations*[6]. One of the authors has shown,

[6] According to [LS88] "A computation is oblivious if its data motion and the operations it executes at a given step are *independent* of the actual values of data."

in [Sin93], that the class of *asynchronous iterative algorithms*, tolerates Slow memory. This class of algorithms can be used to solve problems such as simultaneous linear equations (section 5 below) and finding shortest paths in graphs [Roo93]. Causal memory, which is not supported in Mermera, can run additional applications, including the travelling salesman problem, and the dictionary problem [AHJ90].

Mermera allows the programmer to mix different memory behaviors during the same computation, simply by invoking the appropriate operation to access different memory locations at different times. We provide four kinds of write operations to shared memory: *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 different processes and this order is consistent with each process' program and with the information flow through weaker writes. All *PRAM_Writes* and *CO_Writes* by the same process are observed by all processes in the order they were submitted by 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. The general rule is that if a process i reads a value written by process j then process i must be aware of all previous (in j's program ordering) writes by j that are at least as strong as the write that is being read. The only exception to this rule is when i reads a *Local_Write* by j; it is allowed to have missed previous *Slow_Writes* by j, since they do not have to be reliable.

Liveness. The above description 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 now impose the condition that all *CO_Writes* and *PRAM_Writes* be *eventually* propagated[7] to all processes.

[7]We will not dwell on the exact mechanism of propagation in this paper. It could be done using update messages or invalidate messages.

We do not impose any such condition on *Slow_Writes*. This is because losing any *Slow_Write* does not constrain future writes of any type. On the other hand losing a *PRAM_Write* will cause all subsequent writes (*PRAM_Write* and stronger) to be blocked because receiving any of the subsequent writes would imply that the receiving process is aware of the write that was lost.

3 Implementation

In this section we describe an implementation of Mermera on a network of Sun Sparc 1+ workstations running Unix. Our algorithms rely on full replication, and are update-based, *i.e.*, each process has a copy of the shared memory and this copy is updated as writes occur[8]. A read operation returns the value in the local copy. A write operation updates the local copy and propagates the value to other processes, using version 2.2.5 of the Isis toolkit of multicast protocols [BSS91, BJ87]. The exact manner of transmission depends on the type of the write operation.

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 transmitted on a best-effort basis, *i.e.*, the system tries to propagate them but no guarantees are given[9]. 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.

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.*, the order

[8]The performance impact of this design choice is discussed in section 4.

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

in which these messages are delivered is the same for all processes. This is exactly the property we want for *CO_Write*. The *fbcast*() messages obey a weaker constraint: messages sent by the *same* process are delivered to all processes in the order they were sent. However, *fbcasts* sent by different processes may be interleaved in different orders at different recipients. This suffices for *PRAM_Writes*, so we use *fbcasts* to propagate them. No ordering constraints are guaranteed among *mbcast*() messages[10]. We use this primitive for propagating *Slow_Writes*.

Isis offers no ordering guarantees for messages sent using different primitives, *e.g.*, if two messages are sent one after the other using *abcast*() and *fbcast*(), respectively, they are not necessarily delivered in the order they were sent. Our implementation enforces this ordering explicitly via sequence numbers, and careful control of the order of processing different writes that are batched together.

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 procedure that its handler will execute.

We now explain how we combine the different memory operations. These algorithms are given in figures 2, 3 and 7. All writes other than *CO_Writes* may be buffered. The structure of the buffer is shown in the appendix. Its size can be dynamically adjusted. 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 delivered to, and processed by, its own process. This ensures that the *CO_Write* is applied 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. It is also broadcast

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

```
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 2: Pseudo C code for write operation macros.

when a preset timeout expires. If the buffer is not full then the task issuing the *PRAM_Write* or *Slow_Write* can continue immediately after the local copy has been updated.

The protocol used to broadcast a buffer depends on the type of the strongest write in it, 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*. As discussed in section 7 we may switch in the future 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 (see appendix) 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 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 also be

```
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 3: Pseudo C code for *broadcast*$(\cdot)$.

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 abcast total order) 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 send order from the same source) buffer have been applied, then its updates are ignored.

4 Performance

We measure average *access* and *completion* times for individual Mermera operations. The *access time* of an operation is defined to be the duration of time between its invocation and return. In case of non-coherent write operations, the operation return does not imply that the value written has been propagated to all processes sharing memory. To measure the time taken for the operation

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

to be invoked *and* the propagation to complete we use a metric that we call *completion time*. This is the time taken for each process to execute 100 writes in parallel and for all $100p$ values to reach the p participating processes.

4.1 Measurement Set-up and Method

Our experiments are conducted on a dedicated network of six Sun Sparc 1+ workstations. The parameters we varied are: the number of processes[12] 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). For each parameter setting we run the experiments over 100 times and we report the fastest times measured.

The amount of work done for non-coherent writes may be different each time the operation is invoked. This is because if a write causes the buffer to get full then the propagation of the buffer has to be initiated before the writing process can continue. So, we measure the access time for 100 writes for each setting of the parameters. The access times observed by different processes may be different. This is especially true for *CO_Write* which requires the total ordering provided by *abcast*. Isis achieves this total order by designating one of the participating processes, p_0 to be the sequencer. Consequently, *CO_Writes* by process p_0 have a much faster access time than those by other processes because they do not need any remote communication to determine their position in the total order. The access times shown in table 1 represent the averages of those measured at all processes.

To measure the completion time, each process performs 100 writes between two barrier synchronization calls. The `timeofday` timer, which has 1μs resolution, is sampled after the first barrier, and then again after the second barrier. Clearing the second barrier has the side-effect of guaranteeing that all outstanding writes executed before it will have been received and applied by all processes. The completion times clocked at all the processes are averaged and reported in table 2.

[12] One process per processor.

Procs	CO_Write	PRAM_Write				Slow_Write			
		Buffer sizes				Buffer sizes			
		1	10	100	1000	1	10	100	1000
1	1 330	469	59	12	7	469	55	12	6
2	5 320	1 080	168	25	7	964	161	24	6
3	10 050	1 013	266	39	7	806	259	39	6
4	20 490	1 108	281	52	7	868	278	51	6
5	35 610	1 178	277	63	7	864	272	63	6
6	53 780	1 205	280	71	7	914	271	71	6

Table 1: Write access time in microseconds, averaged over 100 writes and over all processes. Buffer size is given in terms of the number of write operations.

Procs	CO_Write	PRAM_Write			Slow_Write		
		Buffer Sizes			Buffer Sizes		
		1	10	100	1	10	100
1	1 368	504	94	48	468	90	47
2	10 300	2 672	542	262	2 453	539	271
3	15 740	4 129	1 000	524	3 834	997	539
4	28 890	6 380	1 480	858	6 014	1 458	857
5	46 450	9 638	1 884	1 249	8 719	1 879	1 254
6	67 610	12 040	2 479	1 709	12 230	2 468	1 740

Table 2: Write completion time in microseconds, averaged over 100 writes and over all processes.

4.2 Analysis of Results

The discussion in this section centers on write operations. Read operations perform exactly as local virtual memory reads do, which is to say that a read should take 100-200 nanoseconds to complete. We find that write operations vary in their access time between 6 μs for non-coherent ones to 53 ms for coherent write. The gap becomes much smaller, but still very significant, when we consider completion time instead of access time. Non-coherent writes have completion times that are roughly 20-40 times smaller than coherent write. A detailed interpretation of our measurements follows.

In table 1, the case for a buffer size of 1000 is one in which no communication happens, since all 100 writes fit in the buffer. In that case, a *PRAM_Write* or a *Slow_Write* takes 6-7 μs to return, showing that Mermera's overhead aside from communication is negligible.

Buffer size has no effect on the performance of *CO_Writes* because no buffering is done for them. For *PRAM_Write* and *Slow_Write* buffer size is the most important factor that affects access and completion times. This is because buffer size determines the number of messages that are sent and received. Access time for non-coherent writes is almost completely independent of the number of processes because the broadcast of their writes is done asynchronously. We do not know why this is not true for a buffer size of 100.

On the other hand, a task doing a *CO_Write* has to wait till the message is delivered to the writing process before it can continue. The message will be delivered locally only after its position in the total order of *abcasts* is determined. When the task goes into a wait state messages sent by other processes can be received causing an additional delay in the resumption of the waiting task, that is proportionate to the number of processes. This explains the faster than linear (in the number of processes) growth of the access time of *CO_Writes*.

Completion time includes the time spent in doing the asynchronous broadcasts and the time spent in executing tasks that are spawned to process incoming updates. Thus, the number of messages sent and received is a significant determining factor for completion time. This number depends on the buffer size and on the number of processes. In particular, small buffer sizes cause the generation of a large number of small messages. Isis, however, coalesces these small messages into larger messages for more efficient transmission. This explains the observation that, for a given number of processors the completion time grows at a rate *sub-linear* in the number of messages sent.

Access and completion times for *CO_Writes* are close[13] 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 its 100 writes it would have already processed many of the writes by other processes.

The most important conclusion to draw from table 2 is that the completion times for *CO_Write* are 20-40 times slower than for non-coherent writes. One may argue that this comparison of the completion 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

[13]In general, *CO_Write* access times are consistent with the performance of Isis *abcast* for small packets [BSS91].

based schemes (see [CKA91] for an example) exist. Our response to this argument is that there are some applications (*e.g.*, the linear solver of section 5) which intrinsically generate a message traffic that is 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, memory management techniques that benefit coherent memory may also benefit non-coherent memory.

The performance of *PRAM_Write* and *Slow_Write* are comparable. We expect the performance of *Slow_Write* to improve significantly if the propagation is done on a best effort basis rather than in a reliable manner as is done in our current implementation.

5 Solving a System of Equations

In this section we show how to use all the behaviors of Mermera to solve a system of linear equations using an asynchronous iterative method [BT89][14]. Consider a system $Ax + b = 0$ of linear equations in n variables. To solve it, 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 4.

We partition x among the p processes so that each process does the above iteration for x_i in its partition. The vector x is in shared memory while copies of A and b are stored 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 *if* the value is propagated, then it will be observed in the proper order. It may be propagated to some or none of them. A call to *isis_accept_events()* enables pending updates from other processes to be applied to the local copy of shared memory and also for events like timeouts to be handled.

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

```
Epsilon = 0.0001                    /* Desired average accuracy per component of x */
do
{ do
  { AbsoluteDiff = 0;
    for (i = MyLow; i < MyHigh; i++)
    { NewXi = -(b_i + sum_{j=1}^{i-1} a_ij × x_j + sum_{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();                   /* Enable processing of incom-
                                                 ing updates */
    }
  } while ( AbsoluteDiff < Epsilon * (MyHigh - MyLow) )   /* 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 4: 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.

The inner loop is executed until a *local* termination condition is satisfied. When a process reaches local termination it performs a barrier synchronization with other processes. This ensures that each process satisfies its local termination condition and that the components of x that caused the local terminations to succeed, are propagated to all processes. Then each process performs a global termination check by running an iteration to compute *all* the elements of x. If the check succeeds, the program terminates with process 1 writing the solution to a file.

Solver Performance

The program solves a system of randomly generated[15] equations in 1000 variables. The matrix A is dense. The number of processes is varied from 1 to 6. Each process has a copy of A and b. The vector x is in shared memory. The performance is measured in terms of the time it takes for the iterations to converge and for convergence to be detected. This time is called the *convergence time*

[15] To ensure correct convergence, all elements of A are generated smaller than 1 in magnitude, and to guarantee the numerical stability of the algorithm the diagonal elements of A are scaled to be much larger than the other elements.

Procs	T_{CO}	T_{mixed}	T_{CO}/T_{mixed}
1	158.2	157.1	1.0
2	560.5	50.8	11.0
3	296.5	46.4	6.4
4	260.0	37.7	6.9
5	176.3	32.9	5.5
6	201.8	30.0	6.7

Table 3: Solution times in seconds for $n = 1000$. T_{CO}, T_{mixed} are the convergence times using coherent writes only, and using the mixed behavior shown in figure 4, respectively. A purely sequential Gauss-Seidel iteration that does not use Mermera converged in 143 seconds after doing 42 iterations.

and it does not include the time needed for the propagation of A and b to all processes.

A number of buffer sizes between 1 and 1000 are used to determine the effect of different buffer sizes on the performance of the solver. The performance of the program in figure 4 is compared to that of a program in which all *Slow_Writes* are changed to *CO_Writes* and the *PRAM_Writes* are deleted. The fastest convergence times from tens of runs for each parameter setting are used to derive our conclusions. Because of the asynchronous method the number of iterations done by each process varied from run to run. Our measurements are summarized in table 3 and plotted in figure 5. The fastest convergence times regardless of the buffer size are shown.

The main conclusions we derive from these measurements are:

1. Employing non-coherent behavior instead of coherent behavior alone improves the performance of our asynchronous iterative algorithm by a factor ranging from 5.5 to 11.0. From the trend of our measurements we expect this improvement to increase as more processors are added.

2. The best convergence time for the parallel program that uses coherent memory only (176 seconds on 5 processors) is slower than a sequential Gauss-Seidel iteration, which takes 143 seconds.. On the other hand, allowing the parallel program to use non-coherence enables it to beat the sequential one, even with just two processors.

3. The parallel speedup of the program using coherent behavior breaks down at 5 processors while the performance of the program with non-coherent behavior continues to improve up

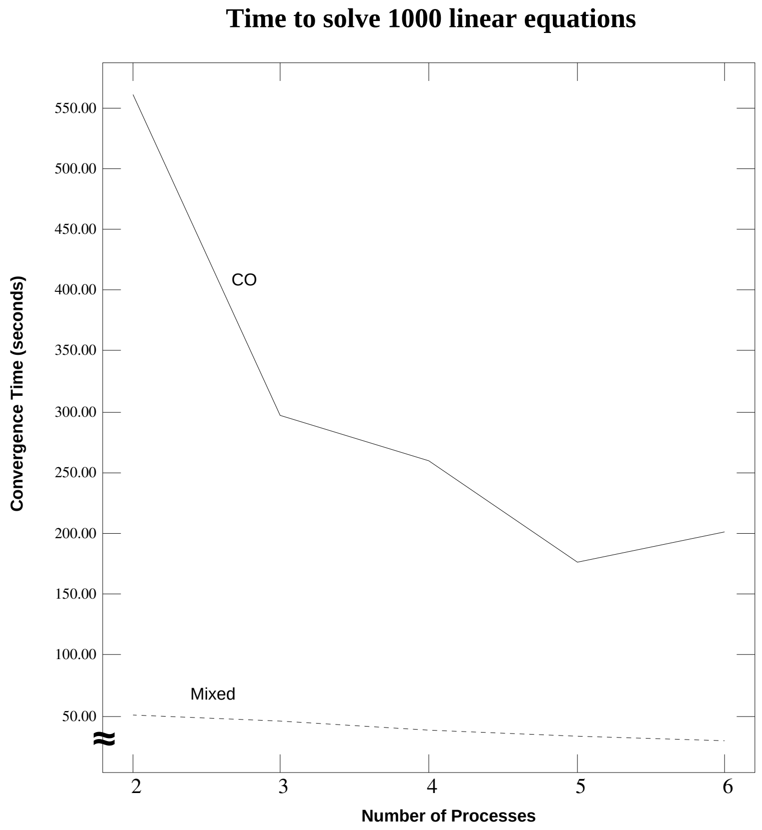

Figure 5: Convergence time vs. number of processors.

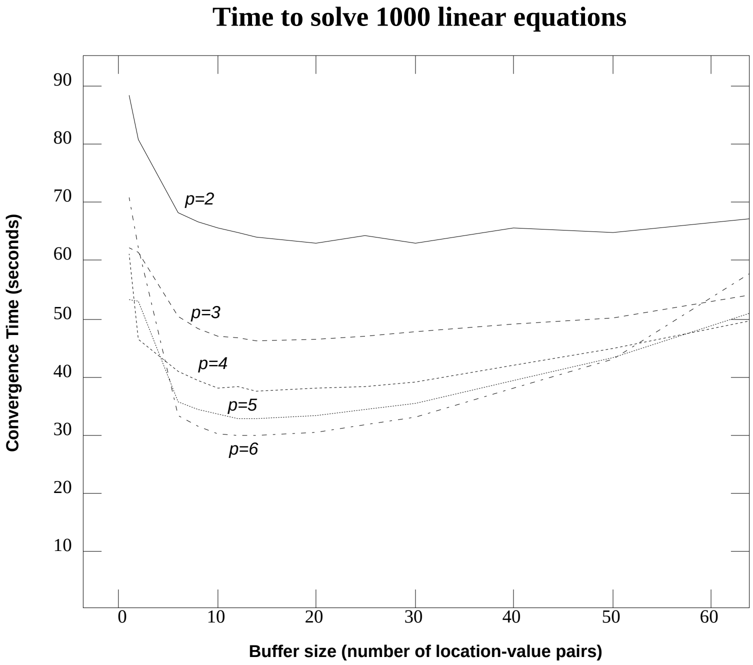

Figure 6: Effect of buffer size on performance.

to at least 6 processors.[16]

Effect of Buffer Size. The specification of Mermera imposes no constraints on the number of non-coherent writes that can be buffered. The implementation is free to choose this parameter. Our observations, summarized in figure 6 show that the buffer size has a significant effect on the performance of the solver. Regardless of the number of processors, performance is best when the buffer size is 12-14. This is because for smaller buffers the frequency at which messages are sent is high which imposes a high overhead of sending and receiving messages on the CPU. If the buffer is large then the frequency of messages is low but each process uses less recent values of the components of x being computed by other processes. The case of 2 processors is an exception where the best performance is achieved when the buffer size is 500.[17]

[16] Because of limited equipment we could not increase the number of processors beyond 6.

[17] For presentation reasons we have excluded that data point from figure 6.

6 Impact of Isis

Our implementation on a network of workstations uses the Isis toolkit to propagate the effect of writes of each process to all other processes. The overhead imposed by our code over and above that of Isis is very small.

For our purposes the following features that are not currently offered by Isis would be useful.

Unreliable, unordered multicasts: Our implementation of *Slow_Writes* uses the *mbcast* multicast primitive provided by Isis. Isis uses a transport layer that guarantees the delivery of all *mbcast* messages in the order they were sent by the sender. Neither of these properties (reliability and FIFO ordering of messages) is required by *Slow_Writes*. We expect *Slow_Write* implemented using an unreliable, unordered multicast to perform better than the current implementation.

Use of IP Multicast: The current version of Isis implements a multicast to a group by sending a point-to-point message to each member of the group. This causes the CPU overhead for sending a message to be paid for each member of a group. The Deering multicast protocol [DC90] can use the multicast facilities provided by the hardware. As a result, the CPU overhead for a multicast can be greatly reduced. An added benefit is that the elapsed time for a multicast is reduced. All writes will benefit from this enhancement and this will also result in a better utilization of the network bandwidth because there will be fewer messages on the network for each write done by any process. The Horus project [vRBC$^+$92] addresses this issue.

Our approach in this implementation was to use Isis multicast 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.

Isis' implementation of *abcast* uses one of the processes as a sequencer. A consequence of this is that coherent writes done by that process have a much faster access time than coherent writes by other processes. This can lead to load imbalances, *e.g.*, in the linear solver the sequencer process went through several more iterations than other processes. To understand this effect we ran the solver with the sequencer process not participating in the pool of processors that perform the iterations. The performance of this version of the program was inferior to that of the version in which the sequencer participated in the iterations. The effect of the load imbalance caused by the fast access time of the sequencer is not clearly understood.

7 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 a completion time that is up to 40 times faster than coherent writes. This advantage in raw write performance translates into an advantage for the equation solver application of a factor of 5-11 in convergence time. The difference arises from two factors: First, coherent and non-coherent reads perform identically in our implementation, and the equation solver performs $n = 1000$ times as many reads as it does writes. Other implementations that replicate the shared memory only partially could give non-coherent reads an advantage over coherent ones. Second, application programs will spend time doing local computation whose performance is independent of that of the shared memory. The equation solver performs $n = 1000$ multiplications and additions for every write to shared memory.

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.

[AHJ90] Mustaque Ahamad, Philip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. Technical Report GIT-CC-90-49, Georgia Institute of Technology, School of Information and Computer Science, 1990.

[Bau78] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 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.

[BR90] Roberto Bisiani and Mosur Ravishankar. Programming the PLUS distributed-memory system. In *Proc. 5th Distributed Memory Computing Conference*, 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 B. Carter, John K. Bennett, and Willy Zwaenopoel. Implementation and performance of Munin. In *Proc. 13th ACM Symp. on Operating System Principles, Asilomar, California*, pages 152–164, Oct. 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.

[DC90] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Trans. on Computer Systems*, 8(2), May 1990.

[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, D. Lenoski, J. Laudon, P. Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 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 Georgia Institute of Technology 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, Boston University, Computer Sciene Dept., May 1992.

[HS93] A. Heddaya and Himanshu Sinha. Computing with non-coherent shared memory. Technical Report TBA, Boston University, Computer Sciene Dept., Feb. 1993.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sep. 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.

[LLJ+93] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: logic overhead and performance. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):41–61, Jan. 1993.

[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/DCS/TR-715, Yale Univ., Dept. of Computer Science, 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.

[Roo93] Nicholas M. Roosevelt. An implementation of mermera on a thinking machines cm-5. Master's thesis, Boston University, Computer Sciene Dept., June 1993.

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

[Sin93] Himanshu Sinha. MERMERA*: Non-coherent Distributed Shared Memory for Parallel Computing*. PhD thesis, Boston University, Computer Science Department, 111 Cummington Street, Boston, MA 02215, May 1993.

[vRBC+92] Robbert van Renesse, Ken Birman, Rober Cooper, Bradford Glade, and Patrick Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures, Seattle, Washington*, pages 269–283, April 1992.

[WW90] W. E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proc. IEEE Symposium on Parallel and Distributed Systems, Dallas, Texas*, Dec. 1990.

Appendix: Memory Update Algorithm

The following table shows the structure of the buffer which is broadcast.

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**
...	...
...	...

The code that applies the location-value pairs received in buffers is given in figures 7 and 8.

```
WaitList[];          /* WaitList[i] contains the RelSeqNo and the SeqNo of the last buffer pro-
                        cessed from process i and a PendingBufList of buffers that have been re-
                        ceived 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 */
BufferStruct *CurrentBuffer;
{   if ((WaitList[CurrentBuffer.Sender].LastRelSeqNo < CurrentBuffer.LastRelSeqNo) ||
        ((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;
        CurrentBuffer = enabled_Buffer(Sender);}
}
```

Figure 7: Pseudo C Code for the handler for the *update_memory* message. The function *enabled_Buffer*(Sender) is described in figure 8. It returns a pointer to a buffer whose updates can be now be applied.

```
/* Checks if any of the pending buffers can be processed */

enabled_Buffer(Sender)
{   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 AbcastQ 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.}
    }
    return CurrentBuffer;
}
```

Figure 8: Pseudo C code for *enabled_Buffer()* function.