# MERMERA:
# NON-COHERENT DISTRIBUTED SHARED MEMORY
# FOR PARALLEL COMPUTING

Himanshu Shekhar Sinha

*April 1993*

**BU-CS-93-005**

Computer Science Department
111 Cummington Street
Boston, MA 02215
Phone: (617)353-8919
E-mail: `hss@cs.bu.edu`

**Note**

This is the technical report version of my Ph.D. thesis. To save a few trees it uses a smaller font than the thesis and it is single-spaced. The abstract appears on page iv.

BOSTON UNIVERSITY

GRADUATE SCHOOL

Dissertation

# MERMERA: NON-COHERENT DISTRIBUTED SHARED MEMORY FOR PARALLEL COMPUTING

by

**HIMANSHU SHEKHAR SINHA**

B. Tech., Indian Institute of Technology, Kharagpur, 1986
M.A., Boston University, 1989

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

1993

# Acknowledgements

Several people influenced this work directly or indirectly. I am deeply indebted to my advisor, Abdelsalam Heddaya, for his continuous support and encouragement. He was involved at each and every step of this research. I would like to thank Azer Bestavros and Steven Homer for their comments and suggestions. I also thank Joyce Friedman and Sharon Salveter for serving on my committee.

I would like to express my gratitude to the graduate student community of the Computer Science department at Boston University. Bob Carter, Nick Roosevelt and Marwan Shaban gave their comments on earlier drafts of this thesis. Chris Lynch and S. Rajagopalan were wonderful officemates. The folks on Christopher Drive tolerated my intrusions into their offices when I needed a break from the long hours at the terminal.

I thank Regina Blaney and Eileen Grabowski for their help in dealing with administrative matters. Lou Henessy's help in my running the Distributed Computing Laboratory was beyond his call of duty and I thank him for that.

Finally, I would like to thank my father, Sachchida Nand Sinha, and my mother, Manju Rani Sinha, for their support and patience through this long endeavor.

MERMERA:
NON-COHERENT DISTRIBUTED SHARED MEMORY
FOR PARALLEL COMPUTING
(Order No. )

HIMANSHU SHEKHAR SINHA
Boston University Graduate School, 1993
Major Professor: Abdelsalam Heddaya, Assistant Professor of Computer Science

## Abstract

The proliferation of inexpensive workstations and networks has prompted several researchers to use such distributed systems for parallel computing. Attempts have been made to offer a shared-memory programming model on such distributed memory computers. Most systems provide a shared-memory that is *coherent* in that all processes that use it agree on the order of all memory events. This dissertation explores the possibility of a significant improvement in the performance of some applications when they use *non-coherent* memory.

First, a new formal model to describe existing non-coherent memories is developed. I use this model to prove that certain problems can be solved using asynchronous iterative algorithms on shared-memory in which the coherence constraints are substantially relaxed. In the course of the development of the model I discovered a new type of non-coherent behavior called *Local Consistency*.

Second, a programming model, MERMERA, is proposed. It provides programmers with a choice of hierarchically related non-coherent behaviors along with one coherent behavior. Thus, one can trade-off the ease of programming with coherent memory for improved performance with non-coherent memory. As an example, I present a program to solve a linear system of equations using an asynchronous iterative algorithm. This program uses all the behaviors offered by MERMERA.

Third, I describe the implementation of MERMERA on a BBN Butterfly TC2000 and on a network of workstations. The performance of a version of the equation solving program that uses all the behaviors of MERMERA is compared with that of a version that uses coherent behavior only. For a system of 1000 equations the former exhibits at least a 5-fold improvement in convergence time over the latter. The version using coherent behavior only does not benefit from employing more than one workstation to solve the problem while the program using non-coherent behavior continues to achieve improved performance as the number of workstations is increased from 1 to 6. This measurement corroborates our belief that non-coherent shared memory can be a performance boon for some applications.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With the proliferation of inexpensive workstations, the idea of using them collectively as a parallel computer has gained widespread acceptance. As a result, several problems that could only be solved on large supercomputers can now be solved on a collection of less powerful processors. Some approaches [BBD+87, BDG+91] use the message passing paradigm for inter-process communication. Others [LH89, MF89] use the shared-memory paradigm.

In the absence of shared-memory in hardware, the message passing paradigm is amenable to efficient implementation. On the other hand, the shared-memory paradigm is a natural extension of programming a uniprocessor machine. But Lipton and Sandberg [LS88] have shown that in the worst case the access time of *coherent*[1] shared memory is proportional to the worst case communication delay among processes. While the coherent shared memory model gives us ease of programming, its implementations suffer from the performance drawback mentioned above.

In this thesis we present a non-coherent shared memory programming model which tries to capture the best of both worlds. We sacrifice some ease of programming from the coherent shared-memory model to get performance that is closer to the message passing model.

Several non-coherent memories have been proposed [LS88, WW90, HA90]. These memories are expected to perform better than coherent memory because of the weaker synchronization requirements of the operations they provide. The weaker synchronization requirements permit operations to be buffered and communication and computation to be overlapped. However, a uniform formal model in which these non-coherent memories can be described has been lacking. We give a formalism based on partial orders on memory events to describe these non-coherent memories. Although we did not set out to invent a new non-coherent behavior, in the course of the development of our formalism we discovered a behavior which we call *local consistency*. This condition should be satisfied by all shared-memories. We use our formalism to prove that totally asynchronous iterative methods to find fixed points can converge on Slow memory [HA90], a non-coherent memory.

We argue that programmers should be given the choice of several non-coherent behaviors, thereby enabling them to trade off programming simplicity for better performance. Taking

[1] We will formally define coherence in Chapter 2. Intuitively, in coherent shared memory all processes agree on the order of events on memory. Sequential Consistency [Lam79] is an example of coherent behavior and in chapter 2 we show that our notion of coherence is equivalent to Sequential Consistency. Therefore, we use the terms *Coherence* and Sequential Consistency synonymously.

this into consideration we give the specification of our model, MERMERA[2], in which we give the programmer a choice of several hierarchically related non-coherent behaviors and one coherent behavior. We show how these different behaviors can be used in a program by implementing a totally asynchronous iterative method for solving a system of linear equations.

A description of the implementation and the performance of MERMERA on two different platforms[3] is also included in this thesis. To the best of our knowledge this is the first implementation of a system that combines coherence and non-coherence. The asynchronous iterative algorithm mentioned above was programmed on our implementations and its performance is reported. The first implementation is a pilot study on a 45 node BBN Butterfly TC2000 which is a distributed memory machine where the time to access a remote location is 3 times the time to access a local location ($\approx 0.7\mu$seconds for a local access). The access time[4] for non-coherent memory using full replication is at least an order of magnitude smaller than the access time for the coherent memory implementation.

The second implementation is on a network of six workstations connected by an Ethernet. Once again, full replication is used. The access time for non-coherent memory is independent of the number of processes sharing memory, while it grows linearly in the number of processors for coherent memory. We observe that non-coherent *write* operations show a 40-fold improvement in *completion time* (defined in Chapter 5) over coherent writes. The linear solver using MERMERA performs at least 5 times faster on non-coherent memory than on coherent memory. Moreover, the performance of the solver using coherent behavior breaks down at six processors, *i.e.*, it takes more time to solve the equations using six processors than it does using five processors. On the other hand, the solver using non-coherent memory continues to show improvement in performance with up to six processors[5]. The coherent solver never performs better than a sequential solver using the Gauss-Seidel iterative method while the non-coherent solver does.

## 1.1 Related Work

In the message passing programming model (Figure 1.1) each process can access data in its address space using read/write operations. Processes communicate with each other by sending and receiving messages using send/receive operations.

In the shared-memory programming model (Figure 1.2) all processes share the same memory. Inter-process communication is achieved by read/write operations to memory. In Figures 1.1 and 1.2 the **P**'s and the **M**'s denote processes and their address spaces respectively.

We use the term *distributed memory machine* for hardware that consists of several processors each with its own memory. The processor-memory pairs are connected by some network. *Our goal is to give a fast shared-memory interface to a distributed memory machine.*

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

[3] An implementation on a Connection Machine is in progress.

[4] The terms *access time* and *completion time* will be defined in Chapters 4 and 5.

[5] Due to limited equipment we could not measure performance on more than six processors.

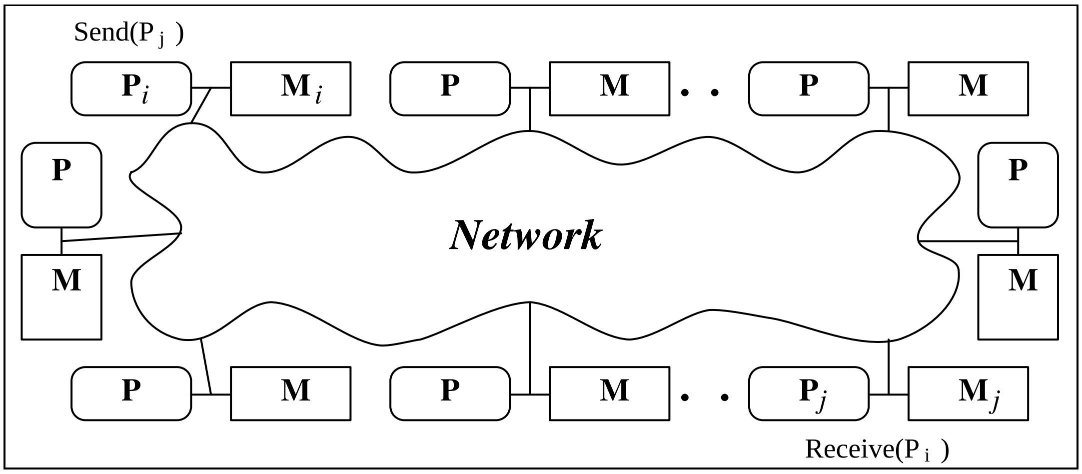


Figure 1.1: The message passing model.

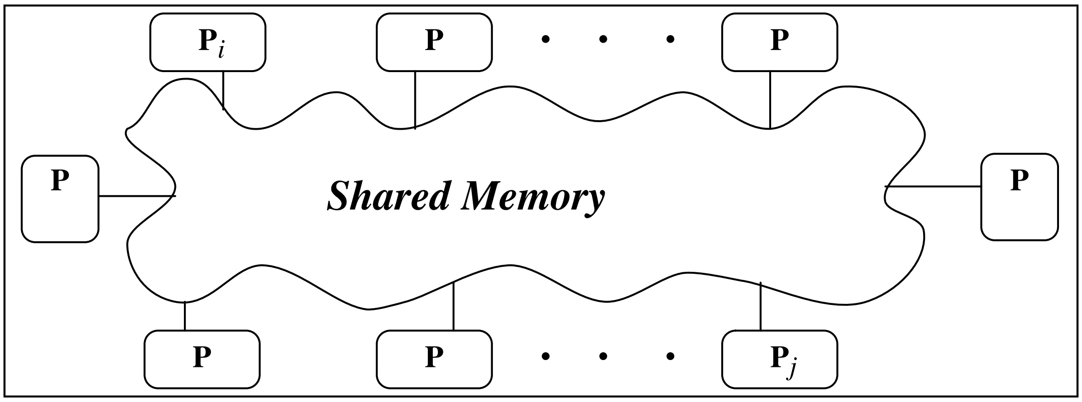


Figure 1.2: The shared-memory model.

The first implementation of the shared-memory model on a distributed system is described in [Li86] and elaborated in [LH89]. It guarantees dynamic atomicity (cf. Chapter 2) which is a stronger condition than sequential consistency [Lam79]. It also uses a page of memory as a unit of data transfer, which can lead to *false sharing* that can further degrade its performance. The distributed shared memory is implemented in libraries linked to the user's program. Mirage [FP89] is another system that provides coherent shared memory. It puts the implementation of distributed shared memory in the kernel.

The first implementation of Mether [MF89] is also a page-based system that guarantees dynamic atomicity. Orca [BKT92] provides coherent shared objects. Their implementation uses full replication, as does ours.

The Pipelined Random Access Memory (PRAM) was proposed by Lipton and Sandberg [LS88]. It guarantees that writes made by the same process are read by all processes in the order they were written by the writer. Writes by different processes can be interleaved in different orders at different processes. A hardware implementation of PRAM is described in [San90, Ser90].

Hutto and Ahamad proposed *Causal* memory, *Slow* memory and *Weak* memory in [HA90]. Causal memory respects the potential causality [Lam78] of memory operations. Slow memory only guarantees that writes by the same process *to the same location* are read by all processes in the order they were written by the writer. Writes to different locations, even by the same process, can be interleaved in different orders at different processes. A design for a Causal memory system is described in [AHJ91] and a formal specification is given in [ABHN91]. We do not know of an implementation of any of these memories.

In Chapter 3 we argue for the combination of different behaviors in one system. Multi-Version memory [WW90] combines the behaviors of Weak memory and Dynamic Atomic memory. An implementation of it was used to speed up a parallel B-Tree algorithm. Later implementations of Mether [MF90, Min91] allow processes to read inconsistent data and the programmer can choose to enforce consistency at any point in the program. The behavior permitted is very similar to that permitted by Multi-Version memory. Attiya and Friedman [AF92] present a correctness condition for a system that supports one coherent behavior and one non-coherent behavior. Their formalism is different from ours and our specification allows a hierarchy of non-coherent behavior.

The architecture community has also moved in the direction of relaxing consistency constraints to address the high latency of sequentially consistent memory. *Processor consistency* [Goo91] is similar to PRAM above with the added condition that all writes to the same location be totally ordered. The PLUS system [BR90] implements processor consistency. Dubois *et al* [DSB86] proposed the idea of relating the ordering of events in the memory to synchronization points in the program. This requires the program to distinguish between ordinary and synchronizing accesses to the memory. Extensions of this idea can be found in [AH90, GLL$^+$90]. *Release Consistency*, defined in [GLL$^+$90] has been implemented in the DASH multiprocessor [LLG$^+$92]. Munin [CBZ91] implements release consistency in software on a network of workstations. Lazy release consistency [KCZ92] is an algorithm for implementing release consistency. Midway [BZ91] uses *entry consistency* in which all shared data is associated with synchronization variables. Here also, the goal is to reduce the amount of communication by propagating data associated with certain synchronization variables.

All these approaches differ from our approach in that they require the programmer to differentiate between synchronization accesses and other accesses. We achieve synchronization through ordinary operations on the memory. Our approach is especially suited for asynchronous algorithms [BT89] that have very weak synchronization requirements.

## 1.2 Outline of this Thesis

In Chapter 2 we present our formalism that describes the different non-coherent behaviors proposed in the literature. It is based on partial orders on memory events. We formally confirm the known hierarchy among the different types of behaviors. Using this formalism we prove that Slow memory, one of the weakest behaviors in this hierarchy, is sufficient for the convergence of totally asynchronous iterative methods [BT89].

In Chapter 3 we argue for the inclusion of different types of behavior in the same system. We extend our formalism to specify the behavior of MERMERA, a system that combines coherence and non-coherence. Our programming model is presented in this chapter. We give a program to solve a linear system of equations using an asynchronous iterative method. This program uses the different behaviors offered by MERMERA.

Chapter 4 describes the implementation and performance of MERMERA on a 45 node BBN TC2000. This was a pilot implementation that allowed us to estimate the performance improvement that could be obtained from non-coherent behavior.

An implementation of MERMERA on a network of Sparcstation 1+s is described in Chapter 5. We use the Isis toolkit [Bir91] for this implementation. We report on the performance of this implementation under a synthetic memory reference pattern and also under the linear equation solver described in Chapter 3. Finally, in Chapter 6 we discuss our conclusions and state some directions for future work.

# Chapter 2

# A Formal Model of Shared Memory

In this chapter we present a formal model of shared memory based on the order of events allowed on the shared memory. We describe several existing memory behaviors using this model. Our formalism enables us to define these behaviors precisely in one model, thereby making comparisons among these behaviors straightforward. Earlier descriptions of these behaviors were in terms of algorithms that implemented these memories, making a formal comparison hard. In the course of our analyses, we also discovered a new type of non-coherent behavior which we call *Local Consistency* (Section 2.2.2).

Section 2.1 introduces the notation we use to describe our model. Section 2.2.1 offers a formal definition of Coherence and shows how it compares with Sequential Consistency—a very commonly used notion of coherence [Lam79]. We prove that both notions are equivalent. Section 2.2.2 formally defines a variety of known non-coherent memories. The relationship between these memories is explored in Section 2.2.3 and a hierarchy is established among them. Finally, in Section 2.3 we use our formalism to prove that Slow memory, one of the weaker memories in the hierarchy, is sufficient for the convergence of the class of totally asynchronous iterative algorithms.

## 2.1 Operations and Orderings

We base our model on memory *events* (*i.e.*, executions of memory operations) rather than on memory states, since the latter are only observable through the former. We describe the correctness conditions of the various kinds of non-coherent memories in terms of the event orderings that they allow.

In this section, we define the essential components that constitute a computation. These components are general enough to enable the formal description of non-coherent memory behavior. Our model consists of a set of processes $P$ sharing a set of memory locations $L$, by executing operations on them. In general, we denote an *operation execution* by $x.o(\bar{a})\!:\!\bar{r}$, where $x$ is the name of the memory location, $o$ stands for the operation (*e.g.*, read or write), and $\bar{a}, \bar{r}$ represent the lists of arguments and results, respectively. An operation execution has an invocation part $x.o(\bar{a})$ and a return part $x.o\!:\!\bar{r}$.

| Notation | Meaning |
|---|---|
| $x.r_i{:}v$ | A *read* event of location $x$, returning $v$ and uniquely identified by $i$. We use $i$ to refer to the event's process. |
| $x.w_i(v)$ | A *write* event that writes the value $v$ to location $x$. |
| $\alpha, \beta \ldots$ | Event variables. |
| $\Delta_i$ | The set of all events in process $i$. |
| $\Delta$ | The set of all events in a computation. $\Delta = \bigcup_i \Delta_i$ |
| $\mathbf{R}_{\ldots}$ | A relation generally on $\Delta$ and generally not transitive. |
| $\mathbf{R}^+_{\ldots}$ | irreflexive transitive closure of $\mathbf{R}_{\ldots}$ |
| $\mathbf{R}_i$ | *Process program ordering* imposed by $i$'s program on its events, $\Delta_i$. Must be a partial order. |
| $\mathbf{R}_{wr}$ | *Writes-to ordering* between write events and the read events that read their values (def. 2.1). |
| $\mathbf{R}$ | Global *program ordering*, the union of all process program orderings and the writes-to ordering. $\mathbf{R} = \bigcup_i \mathbf{R}_i \cup \mathbf{R}_{wr}$. |
| $\mathbf{R}_\Theta$ | Subset of $\mathbf{R}$ that relates only events in $\Theta \subseteq \Delta$. |
| $\Theta^w$ | *write-closure* of $\Theta$, contains $\Theta$ and all write events which are read by events in $\Theta$ (def. 2.5). |
| $\mathbf{R}^w_\Theta$ | $\mathbf{R}_\Theta \cup \{(\alpha,\beta) \mid (\beta \in \Theta) \wedge (\alpha \mathbf{R}_{wr} \beta)\}$, the write-closure of $\mathbf{R}_\Theta$ |

Table 2.1: Summary of notation used in this thesis.

An *event* is a particular operation execution on behalf of a particular process. Events are related to each other by various orderings, such as the ordering induced by each process' program, or the ordering induced by information flow between processes. These orderings are defined as needed throughout this chapter. We denote an event by $x.o_i(\bar{a}){:}\bar{r}$, where $i$ is a unique identifier for the event. With a slight abuse of notation, we will use $i$ to represent the process on behalf of which the event is executed.[1]

In this thesis, we restrict our attention to shared memory that supports only read and write operations.[2] Thus, operation executions can take only the forms of $x.r(){:}v$, or $x.w(v){:}\mathrm{OK}$. For brevity, we will henceforth write $x.r{:}v$ and $x.w(v)$.

A *process* $i$ is a partially ordered set (POSET) of events $\langle \Delta_i, \mathbf{R}_i \rangle$, where $\mathbf{R}_i$ is an irreflexive partial ordering induced by process $i$'s program. The set of all events $\Delta$ is given by $\bigcup_i \Delta_i$.

A glossary of the notation used in this thesis is shown in table 2.1.

To simplify our presentation, we assume, without loss of generality, that

[1]The process identifier is typically encoded in the event identifier. For example, a unique identifier for events belonging to sequential processes can be constructed by appending an event sequence number to the process identifier.

[2]It should be interesting to study shared memory that supports objects with arbitrary operations (*e.g.*, incr/decr, enque/deque), for we can consider the variety of message passing models to be special cases of shared memory in which the locations are queues with various ordering and composition properties.

**Assumption 2.1** *Every value $v$ can be written to each memory location at most once,i.e.,*

$$x.w_i(v_1) \neq x.w_j(v_2) \implies v_1 \neq v_2.$$

This can be easily enforced by appending a logical timestamp to the values written.

The *writes-to ordering*[3] is the last component we need to define our notion of computation. This ordering relates a write event to every other read event that returns its value. Formally,

**Definition 2.1** $\mathbf{R}_{wr} \subseteq \Delta \times \Delta$ *is the minimal relation such that, $\forall v$, for every pair of write and read events in $\Delta$, $x.w_i(v)\, \mathbf{R}_{wr}\, x.r_j{:}v$.*

We can now define a *computation* as follows.

**Definition 2.2** *A computation $E$ is a pair $\langle \Delta, \mathbf{R} \rangle$.*

**Definition 2.3** *We say that a computation is* **well-formed** *if for every read event $\beta \in \Delta$ there exists a write event $\alpha \in \Delta$ such that the value read by $\beta$ is written by $\alpha$, i.e., $\alpha R_{wr} \beta$.*

In this thesis we deal with well-formed computations only.

In the following sections we use the notions of events, program ordering and writes-to ordering defined above, to construct other ordering relations. By constraining these ordering relations in various ways, we arrive at precise formal definitions of a variety of non-coherent shared memory behaviors that have been proposed. We contrast our definitions with those offered in the literature.

## 2.2 Coherent and Non-Coherent Memories

### 2.2.1 Coherent Memories

In this section we describe executions that we consider to be *Coherent*. Intuitively, an execution is Coherent if all processes agree on an ordering of all the events in the execution. After defining coherence formally we will discuss other notions of coherence and relate our notion of coherence to them.

We first define a projection operator $\mathbf{R}(\Theta)$ that projects pairs from $\mathbf{R}$.

**Definition 2.4** $\forall \Theta \subseteq \Delta, \mathbf{R}(\Theta) = \{(\alpha, \beta \in \Theta) | (\alpha \mathbf{R} \beta)\}$.

For notational convenience we sometimes use $\mathbf{R}_\Theta$ to denote $\mathbf{R}(\Theta)$.

We say that a relation constructor $\mathcal{R}$ is *monotonic* iff $\forall \Theta \subseteq \Psi, \mathcal{R}(\Theta) \subseteq \mathcal{R}(\Psi)$.

**Observation 1** *The projection operator $\mathbf{R}(.)$ is monotonic, i.e., if $\Theta \subseteq \Psi \subseteq \Delta$ then $\mathbf{R}_\Theta \subseteq \mathbf{R}_\Psi$ which in turn implies $\mathbf{R}^+_\Theta \subseteq \mathbf{R}^+_\Psi$.*

**Definition 2.5** $\Theta^w$ *is the* write-closure *of $\Theta \subseteq \Delta$ and it is defined as*

$$\Theta^w = \Theta \cup \{\alpha | (\alpha \mathbf{R}_{wr} \beta) \wedge (\beta \in \Theta)\}$$

[3] Inspired by the reads-from ordering used in defining *view serializability* of database transactions [Pap86].

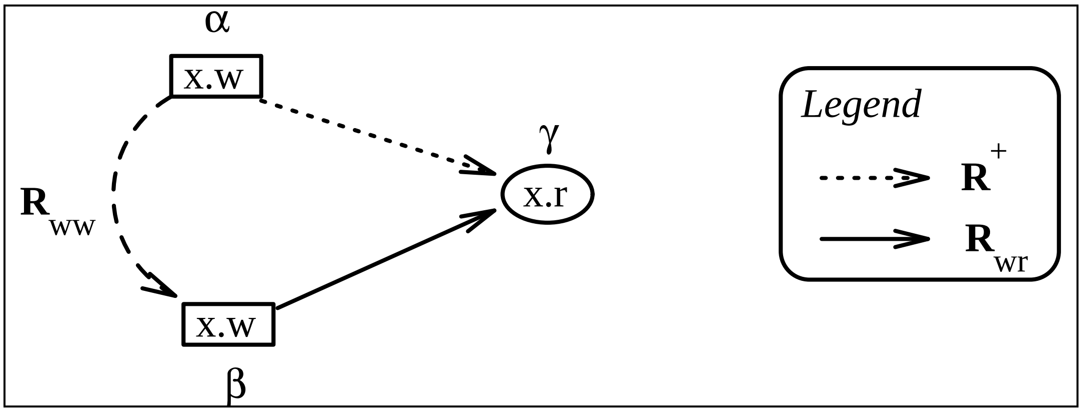


Figure 2.1: An example of $\alpha\mathbf{R}_{ww}(\Theta)\beta$.

**Observation 2** *If* $\Theta \subseteq \Psi \subseteq \Delta$ *then* $\Theta^w \subseteq \Psi^w$.

**Definition 2.6** $\mathbf{R}_\Theta^w = \mathbf{R}_\Theta \cup \{(\alpha,\beta)|(\alpha\mathbf{R}_{wr}\beta) \wedge (\beta \in \Theta)\}$.

$\mathbf{R}_\Theta^w$ is different from $\mathbf{R}_{\Theta^w}$ in that the former does not include any pair consisting of two writes that do not belong to $\Theta$. For example, consider two writes by process $i$, $\alpha,\beta \in \Theta^w$ such that $\alpha,\beta \notin \Theta$ and $\alpha\mathbf{R}_i\beta$. In this case $(\alpha,\beta) \in \mathbf{R}_{\Theta^w}$ but not in $\mathbf{R}_\Theta^w$.

We define a relation constructor $\mathbf{R}_{ww}(\Theta)$, that takes a $\Theta \subseteq \Delta$ and returns a set of ordered pairs of distinct write events from $\Theta^w$ that write to the same location. The relation returned captures our notion of "observing" write events. An event $\alpha$ is **observed** by another event, $\beta$ iff $\alpha\mathbf{R}^+\beta$. $\mathbf{R}_{ww}(\Theta)$ returns ordered pairs of write events according to the order they are observed.

**Definition 2.7** $\mathbf{R}_{ww}(\Theta)$ *is defined on* $\Theta^w$ *as* $(\alpha = x.w_i(v_1))\mathbf{R}_{ww}(\Theta)(\beta = x.w_j(v_2))$ *iff* $\exists((\gamma = x.r_k{:}v_2) \in \Theta)|\ (\alpha\mathbf{R}_{\Theta^w}^+\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma)$.

This relation orders two writes to the same location in the order they are observed by process $k$. Note that not all pairs of write events to a location will be ordered by this relation. $\mathbf{R}_{ww}$ is illustrated in Figure 2.1.

**Lemma 2.1** $\mathbf{R}_{ww}(.)$ *is monotonic, i.e., if* $\Theta \subseteq \Psi \subseteq \Delta$ *then* $\mathbf{R}_{ww}(\Theta) \subseteq \mathbf{R}_{ww}(\Psi)$

**Proof:** Let $(\alpha,\beta) \in \mathbf{R}_{ww}(\Theta)$.

$$\begin{aligned}
&(\alpha,\beta) \in \mathbf{R}_{ww}(\Theta) && \\
&\Rightarrow \exists\gamma \in \Theta|(\alpha\mathbf{R}_{\Theta^w}^+\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma) \wedge (\alpha,\beta \in \Theta^w) && \text{(Def. 2.7)}\\
&\Rightarrow \exists\gamma \in \Psi|(\alpha\mathbf{R}_{\Theta^w}^+\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma) \wedge (\alpha,\beta \in \Theta^w) && \text{(Because } \Theta \subseteq \Psi)\\
&\Rightarrow \exists\gamma \in \Psi|(\alpha\mathbf{R}_{\Psi^w}^+\gamma) \wedge (\beta\mathbf{R}_{wr}\gamma) \wedge (\alpha,\beta \in \Psi^w) && \text{(Observations 1, 2)}\\
&\Rightarrow (\alpha,\beta) \in \mathbf{R}_{ww}(\Psi) && \text{(Def. 2.7)}
\end{aligned}$$

**Definition 2.8** *A computation* $E = \langle\Delta,\mathbf{R}\rangle$ *is* Coherent *iff* $\langle\Delta,(\mathbf{R}\cup\mathbf{R}_{ww}(\Delta))^+\rangle$ *is a partial order.* $\mathbf{R}_{Coherent} = \langle\Delta,(\mathbf{R}\cup\mathbf{R}_{ww}(\Delta))^+\rangle$. *The set of all Coherent computations is denoted by CO.*

There are other notions of coherence prevalent in the literature. A hierarchy of Coherent memories is presented in [HA90]. The most commonly defined notion is that of Sequential Consistency. An execution is Sequentially Consistent if its results were the same 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 [Lam79]. We define the *result* of an execution to be the values read by the read events. We refer to the set of Sequentially Consistent computations by $SC$. However, shared memory implementations like [LH89] and coherent caches in shared memory multiprocessors implement stronger conditions like static atomicity (SA) and dynamic atomicity (DA) [HA90]. In static atomic executions operations take effect for all observers at some specific component event. In dynamic atomic executions operations take effect at any point (in absolute time) during the operation interval as long as the resulting history is equivalent to some serial execution. External consistency (EC) of an execution $E$ requires the existence of a sequential order as in SC but this order is restricted to preserve the order of non-overlapping (in absolute time) operations in $E$. It differs from DA in that an operation can take effect outside the interval (in absolute time) defined by its invocation and its return. The difference between SC and EC is illustrated in Figure 2.2. External Consistency can be used to guarantee progress of information flow between processes but this is not true for Sequential Consistency because a process can continue to read only those values that were written by itself and make progress in executing its program without ever reading values written by other processes. It is possible to construct an equivalent serial execution given such an execution.

The description of Sequential Consistency above allows a process to read values that have not yet (again in absolute time) been written. Hutto and Ahamad define *realizable Sequential Consistency* that excludes such executions.

We observe a relationship between serializability theory [BHG87, Pap86] of transactions and some of these notions of coherence. Among the various notions of serializability are final state serializability (FSR), view serializability (VSR) and conflict serializability (CSR). The two phase locking (2PL) protocol allows a subset of the executions allowed by CSR. If we treat each read or write operation on memory as a transaction then 2PL allows only static atomic executions with the release of a lock being the component event at which an operation takes effect. The notion of External Consistency is the same as that of strict serializability in [Pap86]. VSR in conjunction with Local Consistency (LC) (Def 2.15 below) is the same as Sequential Consistency.

We will now define Sequential Consistency in our notation.

**Definition 2.9** *A computation* $E = \langle \Delta, \mathbf{R} \rangle$ *is* Sequentially Consistent *iff*

1. $\exists$ *a total order* $\mathbf{R}_s$ *over* $\Delta$, *such that* $\langle \Delta, \bigcup_i \mathbf{R}_i \cup \mathbf{R}_s \rangle$ *is a partial order and*

2. $\forall x.w(v_1)\mathbf{R}_{wr}x.r{:}v_1, x.w(v_1)\mathbf{R}_{\mathrm{S}}x.r{:}v_1 \wedge \nexists x.w(v_2)$, *such that* $x.w(v_1)\mathbf{R}_s x.w(v_2)\mathbf{R}_s(x.r{:}v_1)$.

*The set of all Sequentially Consistent computations is denoted by* $SC$.

The first condition states that the sequence defined by $\mathbf{R}_s$ agrees with the ordering of events specified by each process' program. Since $\mathbf{R}_s$ and $\mathbf{R}$ are defined over the same set of

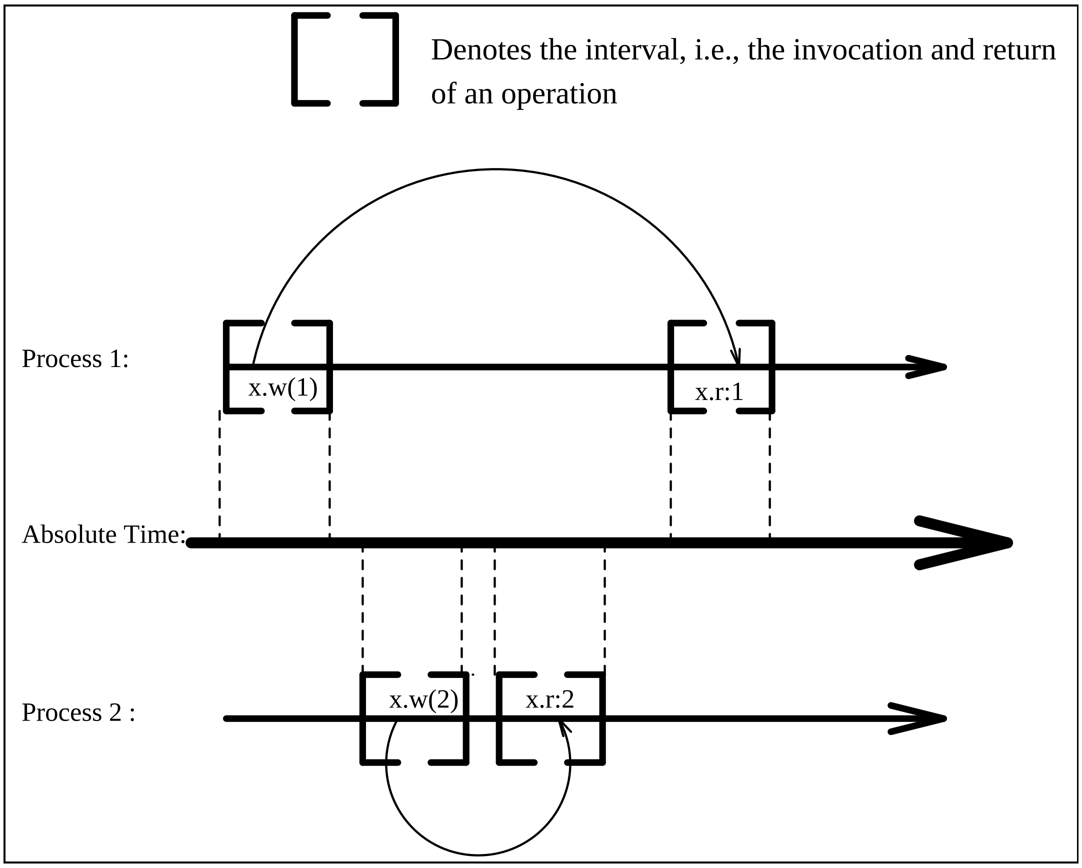


Figure 2.2: An execution that is Sequentially Consistent but not Externally Consistent. It is in SC because both, process 1 and process 2 can agree on the following order: $x.w(1), x.r{:}1, x.w(2), x.r{:}2$. This order is consistent with the order specified by the programs of both processes and each *read* operation returns the value written by the last *write* in the order shown. However this execution is not Externally Consistent because a total order that respects the order imposed by the duration of the events in absolute time does not exist.

events, the values read by corresponding read events in the computations $E = \langle \Delta, \mathbf{R} \rangle$ and $E' = \langle \Delta, \mathbf{R_s} \rangle$ are the same. The second condition states that a read operation returns the value written by the most recent (according to $\mathbf{R}_s$) write in $E'$.

We relate the notion of Sequential Consistency to our definition of Coherence by proving that they are equivalent notions, *i.e.*, they allow the same computations.

**Lemma 2.2** *Any computation, $\langle E, \mathbf{R} \rangle$, that is Sequentially Consistent ($\in SC$) is also Coherent ($\in CO$), i.e., $SC \subseteq CO$.*

**Proof:** We prove our lemma by proving its contra-positive, *i.e.*, by proving that any computation that is not Coherent is not Sequentially Consistent which in turn is proved by contradiction.

Let us assume that, given a non-coherent computation $E = \langle \Delta, \mathbf{R} \rangle$, there exists a total order $\mathbf{R}_s$ on $\Delta$ that satisfies the conditions of Sequential Consistency.

For any computation (Coherent or not) the following observations can be made. From conditions 1 and 2 of the definition of SC,

$$(\alpha, \beta) \in \mathbf{R}^+ \Rightarrow (\alpha, \beta) \in \mathbf{R}_s \tag{2.1}$$

From condition 2 of the definition of SC, and the definition of $\mathbf{R}_{ww}(.)$ (def. 2.7),

$$(\alpha, \beta) \in \mathbf{R}_{ww}(\Delta) \Rightarrow (\alpha, \beta) \in \mathbf{R}_s \tag{2.2}$$

This is because $\alpha \mathbf{R}_{ww}(\Delta)\beta$ requires the existence of a read $\gamma$ that *observes* $\alpha$ but *reads* $\beta$. This implies that $\alpha \mathbf{R}^+ \gamma$ which from predicate 2.1 implies $\alpha \mathbf{R}_s \gamma$. Similarly, $\beta \mathbf{R}_{wr} \gamma \Rightarrow \beta \mathbf{R}_s \gamma$. This establishes that both $\alpha$ and $\beta$ occur before $\gamma$ in the total order defined by $\mathbf{R}_s$. From the second condition of the definition of Sequential Consistency $\beta \mathbf{R}_s \alpha$ is disallowed because it would mean that $\gamma$ did not read the last (in $\mathbf{R}_s$) write to $x$, where $x$ is the location that $\alpha, \beta, \gamma$ operate on. Since $\mathbf{R}_s$ is a total order and $(\beta, \alpha) \notin \mathbf{R}_s$, $(\alpha, \beta) \in \mathbf{R}_s$. Therefore,

$$(\alpha, \beta) \in (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+ \Rightarrow (\alpha, \beta) \in \mathbf{R}_s \tag{2.3}$$

From the definition of coherence we can conclude that if a computation is non-coherent then $\langle \Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+ \rangle$ is not a partial order which implies that there exists a pair of events $\alpha$ and $\beta$ such that both $(\alpha, \beta)$ and $(\beta, \alpha) \in \langle \Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+ \rangle$.

From predicate 2.3 above, this implies that $(\alpha, \beta) \in \mathbf{R}_s \wedge (\beta, \alpha) \in \mathbf{R}_s$ which violates our assumption that $\mathbf{R}_s$ is a total order.

Therefore, if a computation is not Coherent then a total order satisfying the conditions of SC does not exist and a non-coherent computation is not Sequentially Consistent.

To prove the converse, $CO \subseteq SC$, we first state the following lemma which can be proven easily by contradiction.

**Lemma 2.3** *Let $(S, R)$ be a partial order, i.e., $R$ is a partial order on some set $S$. If $p, q \in S$ and $p$ and $q$ are not related by $R$ then $(S, (R \cup (p, q)))$ is a partial order.*

**Lemma 2.4** *Any computation that is Coherent is Sequentially Consistent, i.e., $CO \subseteq SC$[4].*

[4] We are grateful to Trung Dung [Dun92] for this proof.

**Proof:** We prove this by construction. Given a Coherent computation $E = \langle \Delta, \mathbf{R} \rangle$, we will construct a total order $\mathbf{R}_s$ on $\Delta$ that satisfies the two conditions stated in definition 2.9. The steps of the construction are:

1. Let $R_s^0 = (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+$, and $i = 0$.
2. While $\exists (a = x.w(v_1)),\ (b = x.r{:}v_1),\ (c = x.w(v_2)) \in \Delta$ s.t. $a\mathbf{R}_{wr}b$ and $b$ and $c$ are not related in $R_s^i$, do
   (a) $R_s^{i+1} \leftarrow (R_s^i \cup (b, c))^+$.
   (b) $i \leftarrow i + 1$.
3. While $\exists a, b \in \Delta$, s.t. $a$ and $b$ are not related in $R_s^i$ do
   (a) $R_s^{i+1} \leftarrow (R_s^i \cup (a, b))^+$.
   (b) $i \leftarrow i + 1$.

This construction will terminate because $\Delta$ is finite. Let $R_s = R_s^i$, *i.e.*, the relation at the end of the construction. $R_s$ is a partial order because both steps 2 and 3 satisfy the conditions of lemma 2.3. Furthermore, $R_s$ is a total order because step 3 will terminate only after all pairs in $\Delta$ have been related.

We will now show that this $R_s$ satisfies the two conditions required of the total order $\mathbf{R}_s$ in definition 2.9.

1. The first condition of definition 2.9 states that $(\bigcup_i \mathbf{R}_i \cup \mathbf{R}_s)$ is a partial order. This condition is satisfied because we started with $R_s^0 = (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+$, which implies that $\bigcup \mathbf{R}_i \subseteq R_s$.
2. The second condition of definition 2.9 states that
   $\forall x.w(v_1)\mathbf{R}_{wr}x.r{:}v_1\ \nexists x.w(v_2) | x.w(v_1)\mathbf{R}_s x.w(v_2)\mathbf{R}_s(x.r{:}v_1)$.
   Let us assume by way of contradiction that $\exists (a = x.w(v_1)),\ (b = x.r{:}v_1),\ (c = x.w(v_2)) \in \Delta$ s.t. $aR_scR_sb$. Note that $a\mathbf{R}_{wr}b$. There are three cases:
   (a) $(c, b)$ was inserted in $R_s$ in step 1 of the construction above. Then $(c, a) \in \mathbf{R}_{ww}$ (definition 2.7), which means that $(c, a) \in \mathbf{R}_s$ (step 1). But this precludes $(a, c) \in \mathbf{R}_s$ which contradicts our assumption that $(a, c) \in \mathbf{R}_s$.
   (b) $(c, b)$ was inserted in step 2. This is not possible because if $(c, b)$ were not related before step 2 then it would be $(b, c)$ that would be added not $(c, b)$.
   (c) Before step 3 $(b, c)$ would be related so $(c, b)$ cannot be related in step 3.

**Theorem 2.1** *From lemmas 2.2 and 2.4, CO = SC.*

### 2.2.2 Non-Coherent Memories

In this section we discuss computations that violate our definition of Coherence (CO). We consider computations allowed by Causal [HA90] memory, pipelined random access memory (PRAM) [LS88], Slow and Weak memories [HA90] and Multiversion memory (MVM) [WW90]. Our correctness conditions are defined in terms of relations that we construct on a given computation. None of these relations require information that is not already available in a computation denoted by $E = \langle \Delta, \mathbf{R} \rangle$.

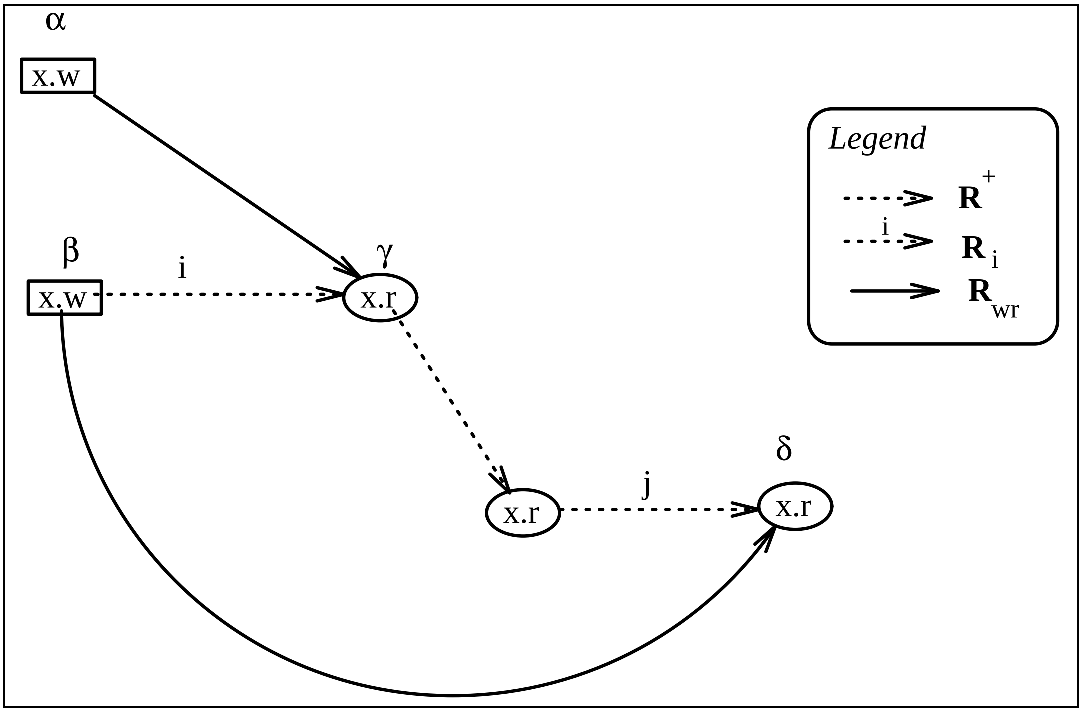


Figure 2.3: Execution that violates overwrite semantics of memory. $\alpha$ and $\beta$ are not related by potential causality but the value that $\delta$ reads has been overwritten (when $\gamma$ read the value written by $\alpha$).

### Causal Memory

In [HA90] Hutto and Ahamad define *Causal memory* to be a memory " ...that requires all processors to agree on the order of causally related effects (writes) but allows events not related by potential causality (*concurrent* events) to be observed in differing orders."

The notion of potential causality is derived from Lamport's definition [Lam78] of the term for message passing systems. In [HA90] a write event is related to a message-send and a read event is related to a message-receive.

This interpretation has two problems. First, it fails to capture the overwrite semantics of memory and permits the computation shown in Figure 2.3 that should not be permitted.

Second, the notion of an order of observation of events is not clear. A more precise definition of Causal memory is presented by Ahamad et al. in [AHJ90].

To give a clear definition of a Causal computation we define two relation constructors $\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ which take a set of events $\Theta \subseteq \Delta$ and return ordered pairs from $\Theta^w$.

Intuitively, $\mathbf{R}_{w/w}(\Theta)$ relates two distinct writes $\alpha$ and $\beta$ to the same location if and only if $\alpha$ follows $\beta$ in $\mathbf{R}^+_\Theta$ but the value written by $\alpha$ is read by a read event that follows $\beta$ (again in $\mathbf{R}^+_\Theta$). It captures the notion of a write event overwriting the value written by previous (in $\mathbf{R}^+_\Theta$) writes.

**Definition 2.10** $\mathbf{R}_{w/w}(\Theta) = \{(\alpha = x.w.(v_1)), (\beta = x.w.(v_2)) | (\alpha, \beta \in \Theta) \wedge \exists \gamma = x.r{:}v_2 \in \Theta | \beta \mathbf{R}^+_\Theta \alpha \mathbf{R}^+_\Theta \gamma\}$

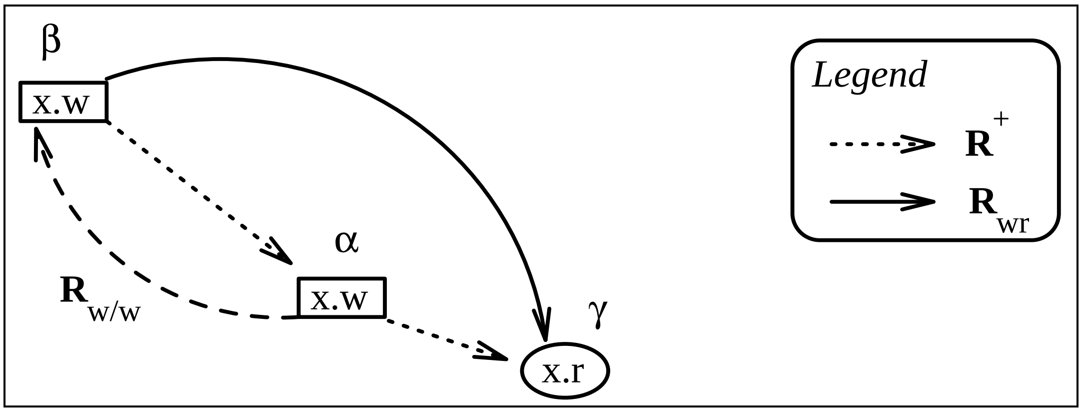


Figure 2.4: $\alpha\mathbf{R}_{w/w}(\Theta)\beta$

Note that $\beta\mathbf{R}_{wr}\gamma$. It is illustrated in Figure 2.4.

**Lemma 2.5** *With $\alpha, \beta, \gamma$ as above, $(\alpha, \beta) \in \mathbf{R}_{w/w}(\Theta) \Rightarrow \alpha\mathbf{R}_{ww}(\Theta)\beta$.*

This follows directly from the definition of $\mathbf{R}_{ww}$. Note that the converse is not necessarily true.

$\mathbf{R}_{r/w}$, illustrated in Figure 2.5 captures a different kind of overwrite semantics than $\mathbf{R}_{w/w}$. While $\mathbf{R}_{w/w}$ relate writes when one of them overwrites the other because of potential causality, $\mathbf{R}_{r/w}$ relates a read with a preceding[5] write that is subsequently read.

**Definition 2.11** $\mathbf{R}_{r/w}(\Theta) = \{(\alpha = x.r{:}v_1, \beta = x.w(v_2)) | (\alpha \in \Theta, \beta \in \Theta^w) \wedge \exists((\gamma = x.w(v_1) \in \Theta^w) \wedge (\delta = x.r{:}v_2) | (\beta(\mathbf{R}^w_\Theta)^+ \alpha \mathbf{R}^+_\Theta \delta))\}$

Note that $\gamma\mathbf{R}_{wr}\alpha$ and $\beta\mathbf{R}_{wr}\delta$.

**Lemma 2.6** *With $\alpha, \beta, \gamma, \delta$ as above, $(\alpha, \beta) \in \mathbf{R}_{r/w}(\Theta) \Rightarrow (\beta\mathbf{R}_{ww}(\Theta)\gamma) \wedge (\gamma\mathbf{R}_{ww}(\Theta)\beta)$.*

**Lemma 2.7** *$\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ are monotonic.*

**Definition 2.12** $\mathbf{R}_{Causal} = \mathbf{R}^+ \cup \mathbf{R}_{r/w}(\Delta) \cup \mathbf{R}_{w/w}(\Delta)$. *A computation is* Causal *iff $\langle \Delta, \mathbf{R}_{Causal} \rangle$ is a partial order.*

In Figure 2.6 we give an example of a computation that is Causal but not Coherent (def. 2.8). It also shows the pairs of events that cause $\langle \Delta, (\mathbf{R} \cup \mathbf{R}_{ww})^+ \rangle$ to not be a partial order. A *memory is Causal* iff it permits Causal computations only.

---

[5] *Precede* and *subsequent* refer to the order imposed by potential causality.

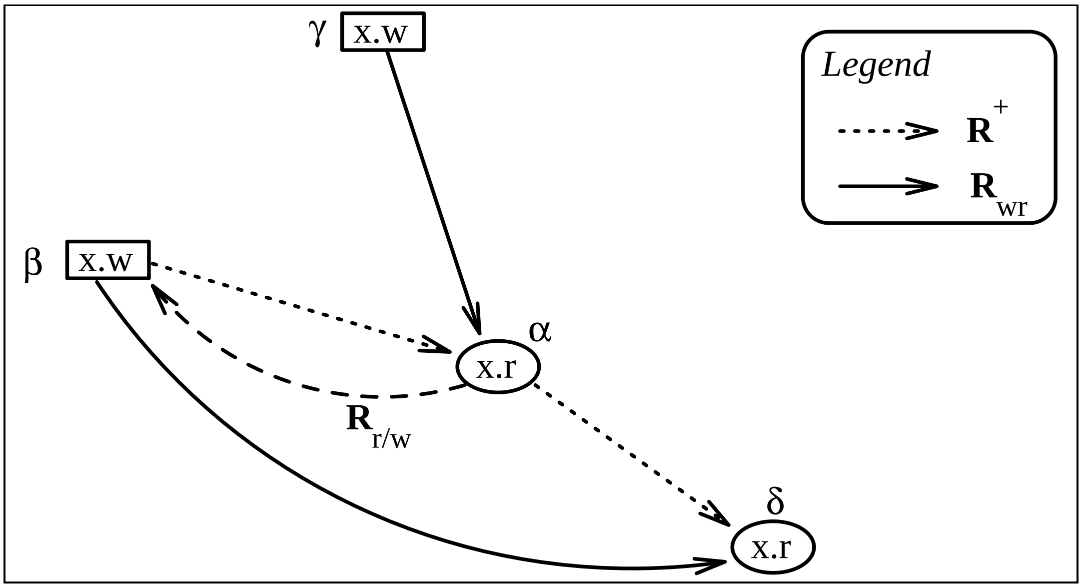


Figure 2.5: $\alpha \mathbf{R}_{r/w}(\Theta)\beta$

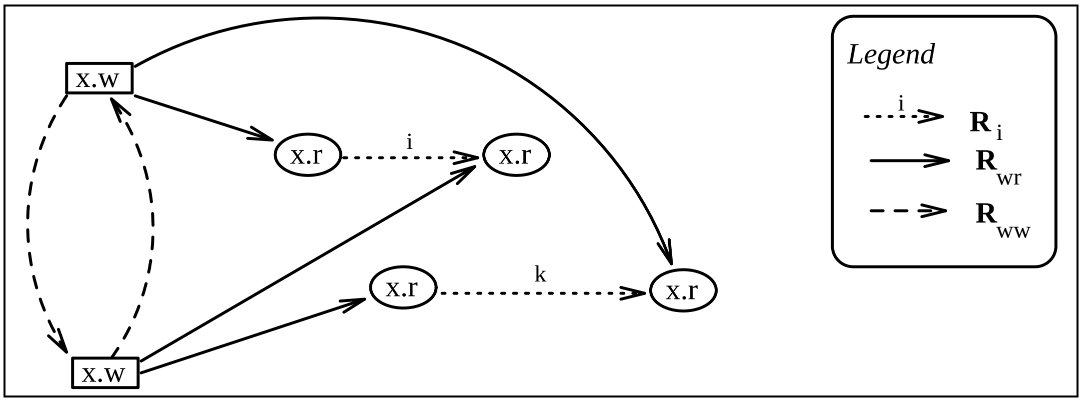


Figure 2.6: A computation that is *Causal* but not *Coherent*.

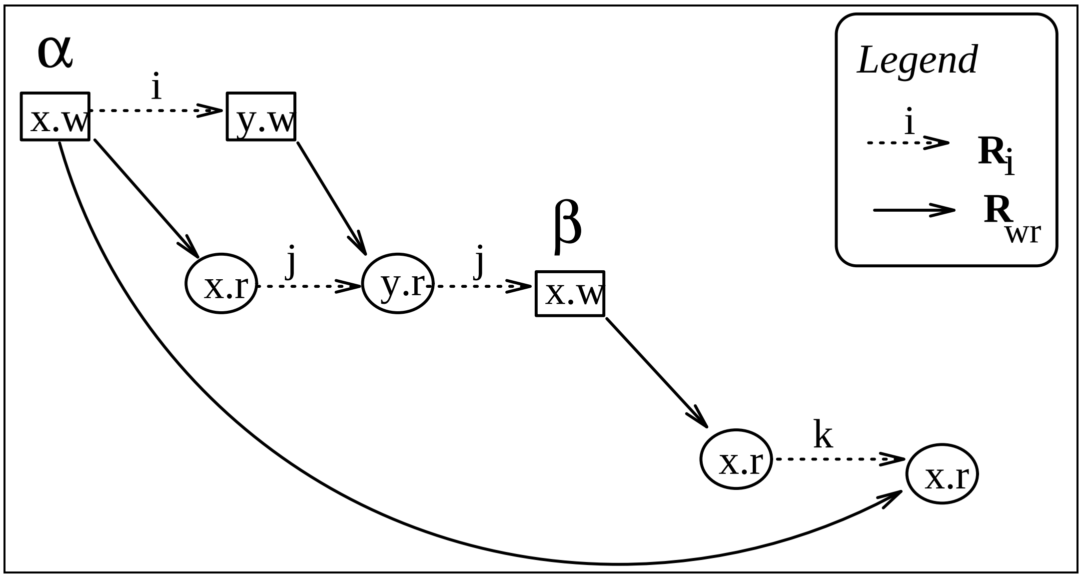


Figure 2.7: A computation that is allowed by *PRAM but* is not *Causal.* $\alpha$ and $\beta$ are causally related but process $k$ observes them in an order that violates this causality.

### PRAM

Pipelined Random Access Memory (PRAM) is proposed in [LS88]. In a PRAM system each process has a copy of every shared location. A process always reads the local copy. It writes to the local copy and broadcasts the value written to all processes. The writing process does not wait for the broadcast to complete. When a process receives the value broadcast it updates its local copy.

To specify the correctness conditions for PRAM computations in terms of ordering of events that are allowed, we use the following relations. Let $\Delta_{ij} = \Delta_i \cup \Delta_j$.

**Definition 2.13** $\mathbf{R}_{PRAM}(\Delta_{ij}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta) \cup \mathbf{R}_{r/w}(\Delta_i))$*, where* $\Theta = \Delta_{ij}$*. A computation is a* PRAM *computation iff* $\forall i, j \in P$*, the set of all processes,* $\langle \Delta^w_{ij}, \mathbf{R}_{PRAM}(\Delta_{ij}) \rangle$ *is a partial order.*

A computation that is a PRAM computation but is not a Causal computation is shown in Figure 2.7. A memory is *PRAM* iff it permits only PRAM computations.

### Slow Memory

Hutto and Ahamad [HA90] define Slow memory as follows: "...reads must return *some* value that has been previously written to the location being read. Once a value has been read, no earlier writes to that location (by the processor that wrote the value read earlier) can be returned. Local writes must be immediately visible. ...".

To describe the computations allowed by Slow memory we define $\Delta_{ijx}$ the set of events pertaining to location $x$ and belonging to process $i$ and process $j$, *i.e.*,
$\Delta_{ijx} = \{\alpha \in \Delta_{ij} | \alpha = x.o\}$.

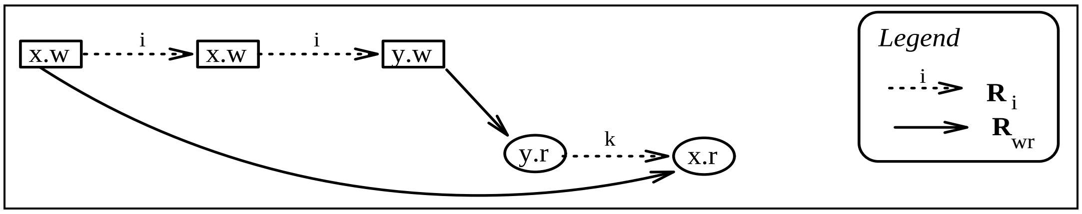


Figure 2.8: A computation that is *Slow* but is not allowed by *PRAM*.

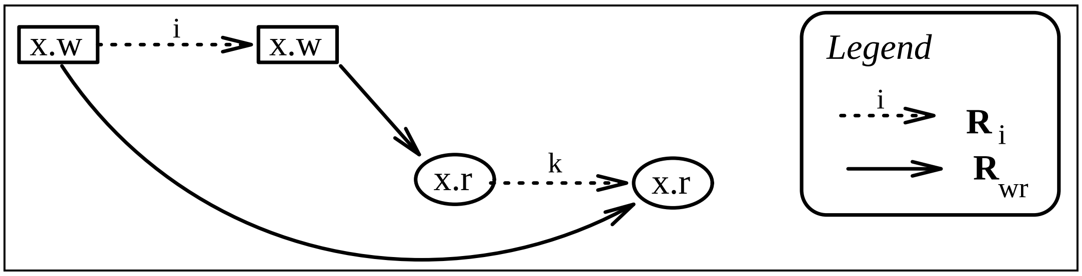


Figure 2.9: An execution that is *Locally Consistent* but not *Slow*.

**Definition 2.14** $\mathbf{R}_{Slow}(\Delta_{ijx}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta) \cup \mathbf{R}_{r/w}(\Delta_{ix})$, *where* $\Theta = \Delta_{ijx}$. *A computation is a* Slow *computation iff* $(\forall i, j \in P \wedge \forall x \in L\langle \Delta^w_{ijx}, \mathbf{R}_{Slow}(\Delta_{ijx})\rangle$ *is a partial order.*

A computation that is Slow but not PRAM is shown in Figure 2.8. A *memory is Slow* iff it allows Slow computations only.

### Locally Consistent Memory

In this section we define a new condition called *Local Consistency* (LC). This condition requires that each process observe events as if they were executed on a single processor. This is different from Sequential Consistency (SC, def 2.9) in that SC requires *all* events to occur as if they were executed on one processor while LC requires that only events observed by a process appear to *it* as if they were executed on a single processor. Different processes may observe the same events in different orders.

**Definition 2.15** $\mathbf{R}_{LC}(\Delta_{ix}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta^w) \cup \mathbf{R}_{r/w}(\Theta))$, *where* $\Theta = \Delta_{ix}$. *A computation is* Locally Consistent *iff* $\forall i, x\langle \Delta^w_{ix}, \mathbf{R}_{LC}(\Delta_{ix})\rangle$ *is a partial order.*

A computation that is Locally Consistent but that is not Slow is shown in figure 2.9. A *memory is Locally Consistent* iff it permits Locally Consistent computations only.

### Weak Memory

The last type of computation we describe is allowed by *Weak* memory of [HA90]. It requires that a read return any value that is previously written by a write operation. The notion of previous is not defined.

**Definition 2.16** *A computation is* Weak *iff* $\forall(\alpha \in \Delta | \alpha.o = read)\exists(\beta \in \Delta)|\beta\mathbf{R}_{wr}\alpha$.

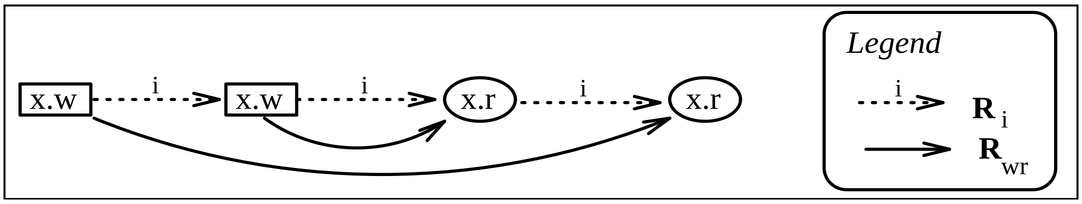


Figure 2.10: An computation that is *Weak* but not *Locally Consistent.*

| Memory Type | Partial Order Relation |
|---|---|
| Coherent | $\langle \Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta))^+ \rangle$ |
| Causal | $\langle \Delta, (\mathbf{R}^+ \cup \mathbf{R}_{w/w}(\Delta) \cup \mathbf{R}_{r/w}(\Delta)) \rangle$ |
| PRAM | $\forall i, j, (\langle \Delta^w_{ij}, ((\mathbf{R}^w_{\Delta_{ij}})^+ \cup \mathbf{R}_{w/w}(\Delta_{ij}) \cup \mathbf{R}_{r/w}(\Delta_i)) \rangle)$. |
| Slow | $\forall i, j, x(\langle \Delta^w_{ijx}, ((\mathbf{R}^w_{\Delta_{ijx}})^+ \cup \mathbf{R}_{w/w}(\Delta_{ijx}) \cup \mathbf{R}_{r/w}(\Delta_{ix})) \rangle)$ |
| LC | $\forall i, x(\langle \Delta^w_{ix}, ((\mathbf{R}^w_{\Delta_{ix}})^+ \cup \mathbf{R}_{w/w}(\Delta^w_{ix}) \cup \mathbf{R}_{r/w}(\Delta_{ix})) \rangle)$ |

Table 2.2: Summary of correctness conditions for different types of memories.

This is the same condition as our well-formedness condition (def. 2.3). An execution that is Weak but not Locally Consistent is shown in Figure 2.10. A memory is Weak iff it permits Weak computations only.

### 2.2.3 Relating Memories

In this section we relate the various kinds of memory we have discussed so far in this chapter. For ease of reference we summarize the correctness conditions of the different types of memories in table 2.2.

We establish parts of the hierarchy shown in Figure 2.11. In the figure, an $A \subset B$ implies that the set of computations allowed by $A$ is a strict subset of the set of computations allowed by $B$.

**Theorem 2.2** *As shown in Figure 2.11,*

$$CO \subset Causal \subset PRAM \subset Slow \subset LC \subset Weak$$

*CO, Causal, PRAM, Slow, LC and Weak each represent the set of all computations allowed by the memory behavior they name.*

**Proof:** From lemma 2.5 and lemma 2.6 we can conclude that if $\mathbf{R}_{Causal}$ is not a partial order then $\mathbf{R}_{Coherent}$ is not a partial order. Therefore any computation that is not Causal cannot be Coherent (*i.e.*, a Coherent computation is Causal). We showed a computation that is Causal but not Coherent in Figure 2.6. Therefore, $CO \subset Causal$.

From the monotonicity property of $\mathbf{R}_{w/w}$ and $\mathbf{R}_{r/w}$ (lemma 2.7) it is obvious that if $\mathbf{R}_{PRAM}(\Delta_{ij})$ is not a partial order for some $i, j$ then $\mathbf{R}_{Causal}$ cannot be a partial order because $\Delta_{ij} \subseteq \Delta$. Therefore any computation that is not a PRAM computation cannot

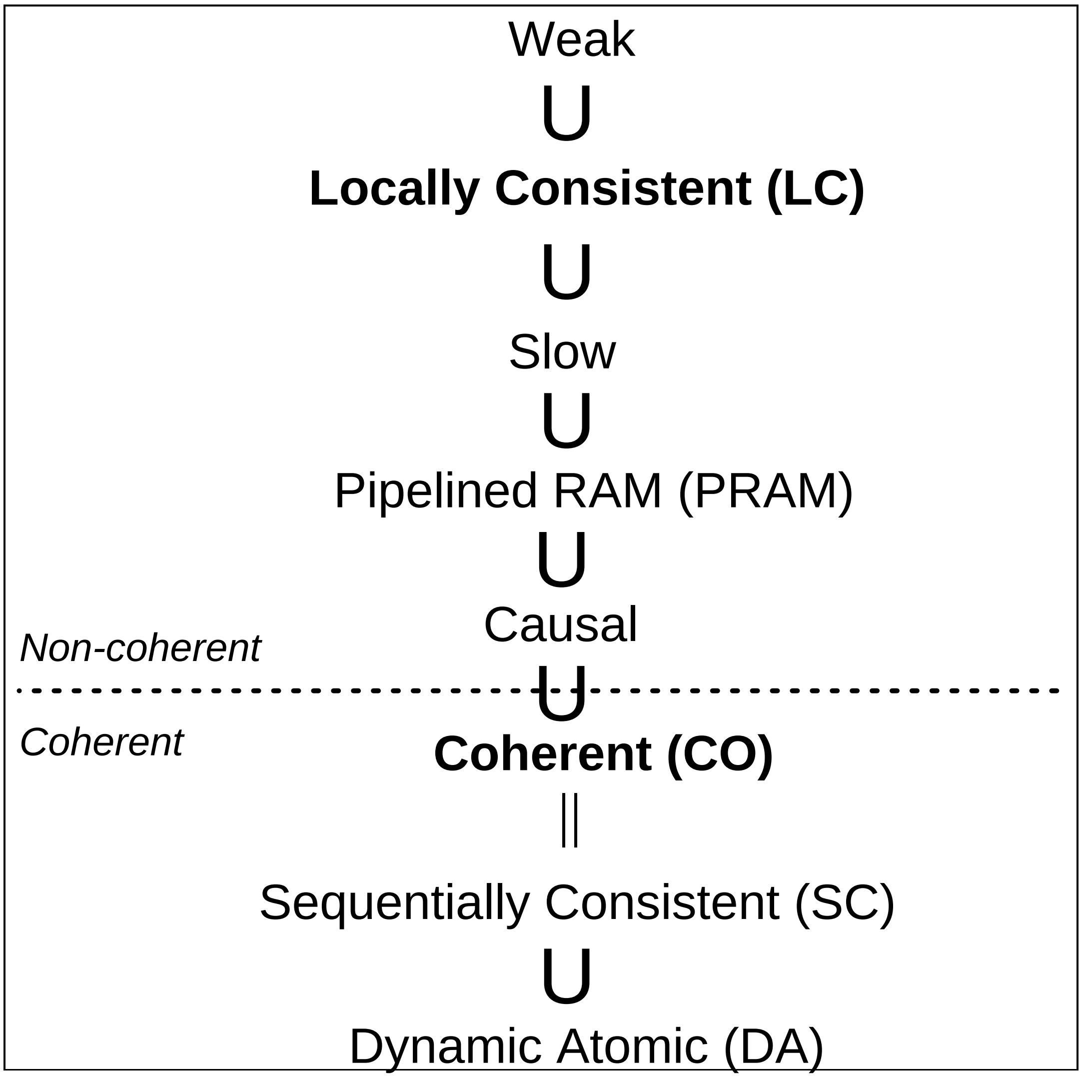


Figure 2.11: Hierarchy of Memories

be Causal (*i.e.*, all Causal computations are PRAM computations too). In Figure 2.7 we showed a PRAM computation that is not Causal. Therefore, $Causal \subset PRAM$.

Since $\Delta_{ijx} \subseteq \Delta_{ij}$ we can conclude from the same monotonicity property as above that if $\mathbf{R}_{Slow}(\Delta_{ijx})$ is not a partial order then $\mathbf{R}_{pram}(\Delta_{ij})$ is not a partial order. Therefore a computation that is not Slow is not a PRAM computation either. In other words all PRAM computations are Slow computations. As shown in Figure 2.8, the converse is not true. Therefore $PRAM \subset Slow$.

The argument to show that a Locally Consistent computation must be Slow is slightly different. The first and third terms in the definition of $\mathbf{R}_{LC}(\Delta_{ix})$ are subsets of the corresponding terms of $\mathbf{R}_{Slow}(\Delta_{ijx})$. The second term in $\mathbf{R}_{LC}(\Delta_{ix})$ is $\mathbf{R}_{w/w}(\Delta^w_{ix})$ while the second term in $\mathbf{R}_{Slow}(\Delta_{ijx})$ is $\mathbf{R}_{w/w}(\Delta_{ijx})$. If there exists $(\alpha, \beta) \in \mathbf{R}_{w/w}(\Delta^w_{ix})$ such that $(\mathbf{R}^w_{\Delta_{ix}})^+ \cup \mathbf{R}_{w/w}(\Delta^w_{ix})$ is not a partial order (implying that the computation is not Locally Consistent) then there are two possibilities. We show that in both cases there will be a $j$ such that $(\Delta^w_{ijx}, \mathbf{R}_{Slow}(\Delta_{ijx}))$ will not be a partial order.

1. $\beta, \alpha \in \Delta_{ix}$ in which case $(\alpha, \beta) \in \mathbf{R}_{w/w}(\Delta_{ijx})$, *or*

2. $\alpha \in \Delta_{ix}, \beta \in (\Delta^w_{ix} - \Delta_{ix})$ in which case, $\exists j | (\mathbf{R}^w_{\Delta_{ijx}})^+ \cup \mathbf{R}_{w/w}(\Delta_{ijx})$ is not a partial order. The $j$ is the process to which $\beta$ belongs.

Therefore, a computation that is not Locally Consistent is not Slow either.

## 2.3 Asynchronous Iterations on Slow Memory

We will now use the formalism that we developed in the last section to show that under certain conditions Slow memory is sufficient for the correctness of *totally asynchronous iterative algorithms* summarized in [BT89, BT90]. In Section 3.3 we show how non-coherent memory can be used to solve a system of linear equations.

Consider an asynchronous iterative algorithm that finds the solution to $x \leftarrow f(x)$ where $x$ is a vector of length $n$. The $i^{th}$ component of $x$ is denoted by $x_i$. For simplicity we assume that we have $n$ processors and that the $i$th processor computes $x_i$.

Let $x_j(t)$ be the value of $x_j$ at time $t$, where $t$ is an integer variable used to index events of interest in the system. $T^i$ is the set of times at which $x_i$ is updated in the following manner.

$$x_i(t+1) = \begin{cases} f_i(x_1(\tau^i_1(t)), x_2(\tau^i_2(t)), \ldots, x_n(\tau^i_n(t))) & \forall t \in T^i \\ x_i(t) & otherwise \end{cases}$$

where $0 \leq \tau^i_j(t) \leq t$. $\tau^i_j(t)$ is an integer that denotes the version of $x_j$ used by processor $i$ at time $t$ to compute $x_i(t+1)$.

It has been shown in [BT90] that if $f$ is a contraction mapping and the implementation of the algorithm to solve the problem satisfies the total asynchronism assumption given below then the iteration described above will converge. The total asynchronism assumption states that

1. the sets $T^i$ are infinite, and,

2. if $\{t_k\}$ is a sequence of $T^i$ which tends to infinity then $\lim_{k \rightarrow \infty} \tau^i_j(t_k) = \infty$.

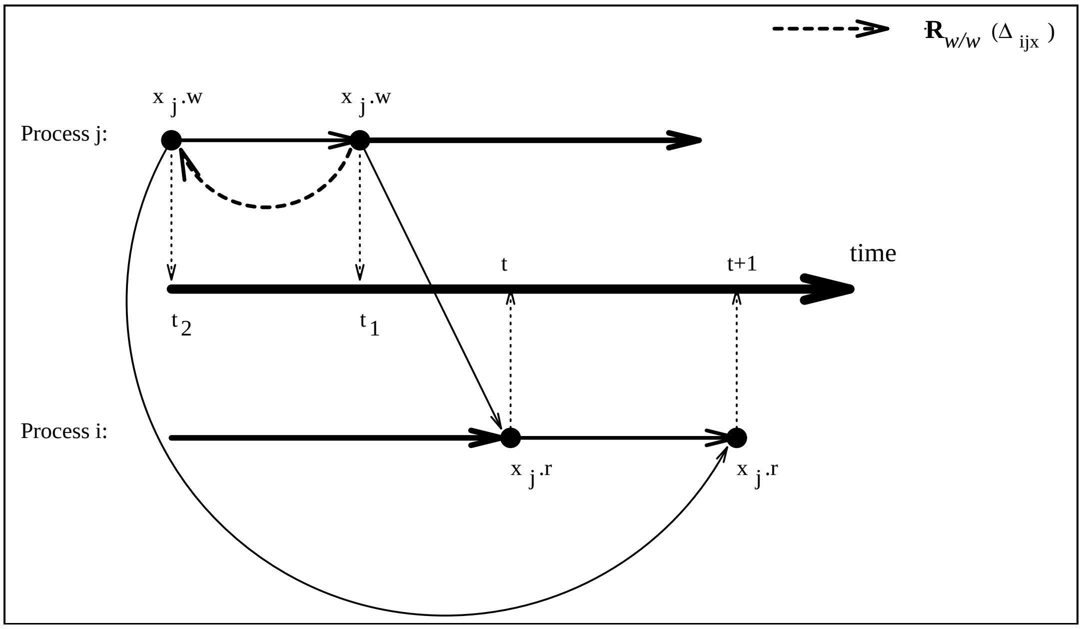


Figure 2.12: A computation in which $\tau^i_j(t) = t_1 > \tau^i_j(t+1)$. In this figure $\tau^i_j(t) = t_1$ and $\tau^i_j(t+1) = t_2$

We will now show that under certain conditions the total asynchronism assumption is satisfied by a system using Slow memory to store $x$.

The first condition, that every set $T^i$ be infinite must be satisfied by the program implementing this iteration. An infinite loop that repeatedly computes $f_i$ satisfies this condition[6].

To prove the second condition we first show that,

$$\tau^i_j(t+1) \geq \tau^i_j(t)$$

Assume, by way of contradiction, that there exist $i, j$ and $t$ such that $\tau^i_j(t) > \tau^i_j(t+1)$. Let $\tau^i_j(t) = t_1$ and $\tau^i_j(t+1) = t_2$, $(t_1 > t_2)$. An execution that allows this will contain the computation segment shown in Figure 2.12. This is not permitted by Slow memory (def. 2.14) because the order imposed by $\mathbf{R}_{w/w}(\Delta_{ijx})$ on the two writes by process j to $x_j$ in Figure 2.12 creates a cycle in $\mathbf{R}_{Slow}$. Contradiction. Thus $\tau^i_j(t)$ is a monotonically non-decreasing function. If an implementation of Slow memory guarantees progress, *i.e.*, a write by a processor, $i$, eventually reaches all other processors, or a subsequent[7] write to that variable by $i$ does, then $\tau^i_j(t)$ is guaranteed to increase. Since $x_j$ is written infinitely often (because of condition 1), $\lim_{t\to\infty} \tau^i_j(t) = \infty$.

A comprehensive list of fixed point problems that converge in a totally asynchronous iteration can be found in chapter 6 of [BT89]. As an example we will use this method to find the solution to a linear system of equations, $x = Ax + b$. Such an iteration will converge if

[6]We will deal with the termination of the iteration in Chapter3.

[7]In $\mathbf{R}_i$.

the spectral radius[8] of $|A|$, $\rho(|A|) < 1$ [Bau78]. Application of this method to optimization problems, the shortest path problem, solution of differential equations and network flow problems is described in [BT89, BT90].

In the next chapter we motivate and propose a system that provides programmers with a choice of non-coherent behaviors that are related by the hierarchy established in theorem 2.2.

[8] The spectral radius $\rho(A)$ of a square matrix $A$ is defined as the maximum of the magnitudes of the eigenvalues of $A$.

# Chapter 3

# MERMERA: A System that Combines Coherent and Non-Coherent Memories

In the last chapter we introduced a formal model for describing different non-coherent behaviors and we established a hierarchy of non-coherent memories. In this chapter we propose our system, MERMERA, which gives programmers a choice of coherent and non-coherent behaviors.

In Section 3.1 we argue for the inclusion of different types of behaviors in the programming model. Section 3.2 specifies MERMERA, a system that provides a combination of the behaviors described in Chapter 2. Our formal model of Chapter 2 is extended in Section 3.2.2 to describe the behavior of MERMERA. Finally, in Section 3.3, we show how the different behaviors provided by MERMERA can be used in one program.

## 3.1 Algorithms that Tolerate Non-coherence

In the last chapter we established the hierarchy shown in Figure 3.1. In the figure, the subset relationship implies that the set of computations allowed by one type of memory is a proper subset of computations allowed by a memory higher in the hierarchy. The question that arises is: *Which of these memories should be provided to programmers?* Memories higher in the hierarchy have weaker ordering requirements making efficient implementations possible. But, not all applications can run on the weaker memories. In the rest of this section we will list the applications known to run on each of the memories in the hierarchy.

Weak memory would be the most efficient to implement but we are not aware of any applications that can run on Weak memory alone. Multi-version Memory (MVM) [WW90] is a memory in which the programmer has access to Weak memory as well as Dynamic Atomic memory. MVM was used to implement B-trees.

In [HA90], Hutto and Ahamad show how a memory that handles exactly one operation at a time (*i.e.*, serially) can be simulated by distributed processors using Slow memory. But we want to focus on classes of programs that can run directly on Slow memory so that one can exploit its *potential* performance advantages. We showed in Section 2.3 that Slow

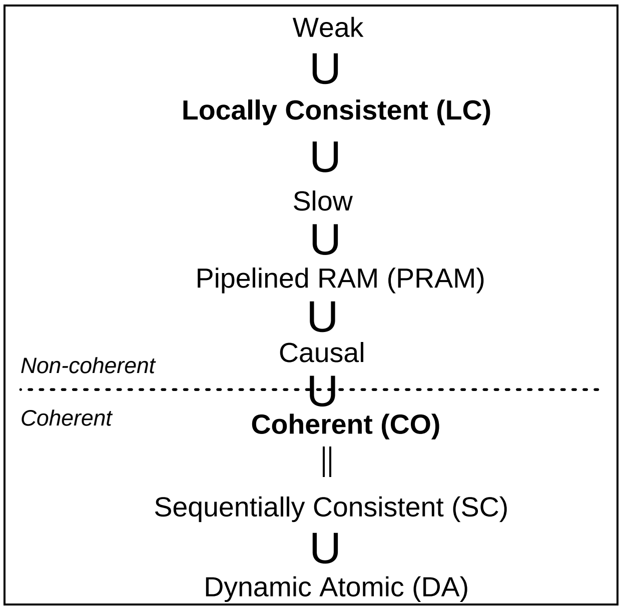


Figure 3.1: Hierarchy of Shared Memory.

Memory is sufficient for the *convergence* of certain asynchronous iterative algorithms to find fix points. But one would need a stronger behavior for *detecting* that the iteration has converged. We believe that asynchronous iterative graph algorithms (*e.g.*, Bellman Ford shortest path algorithm) can also be implemented efficiently on Slow memory.

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*[1]. They also prove that in these computations, whenever the actual computation dominates the synchronization[2] overhead, PRAM is much more efficient than Sequentially Consistent memory.

The use of Causal memory to solve the traveling salesman problem, the dictionary problem and to find the solution of a system of linear equations is described in [AHJ90].

Hutto and Ahamad showed in [HA90] that one cannot achieve mutual exclusion using Slow Memory. Attiya and Friedman [AF92] proved that any solution to the mutual exclusion problem using non-coherent memory will either involve a centralized server or will require the participation of all processes whether they want to enter the critical section or not.

This discussion illustrates that different problems are amenable to efficient solutions on different kinds of memory. For this reason we propose that programmers be given a choice of behaviors. We expect programmers to first program using coherent behavior because of their familiarity with it. They can then trade off the ease of programming for performance by using non-coherent behavior where the program can tolerate it.

[1] "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." [LS88]

[2] Synchronization in the algorithm *not* in the memory.

## 3.2 Specification of Mermera

In this section we briefly describe each of the memory behaviors supported by Mermera and explain how the behaviors are combined. Our system combines the behaviors of Coherent Memory, Pipelined RAM, Slow Memory and Locally Consistent Memory. This choice is rather arbitrary. Our test application could use all of these behaviors and these turned out to be easy to implement with the tools we had. Our model consists of processes sharing a region of their address space. These processes may be running on different processors. We first describe the behavior informally and then we extend our model of Chapter 2 to describe Mermera.

### 3.2.1 Informal Specification

Programs perform *Read* and *Write* operations to the shared memory. We provide four kinds of write operations: *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.

**CO_Write(loc, val):** This operation provides the behavior specified by Coherent Memory (Section 2.2.1), *i.e.*, all *CO_Write* are totally ordered.

**PRAM_Write(loc, val):** This operation provides the behavior specified by PRAM (Section 2.2.2). The order of all *PRAM_Writes* by the *same* process is respected by all processes, *i.e.*, if a process performs two *PRAM_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_Write(loc, val):** This operation provides Slow Memory. All *Slow_Write* by *the same process to the same location* are ordered by all processes in the order they were written. *Slow_Write* to different locations by the same process may be ordered differently by different processes.

**Local_Write(loc, val):** This operation makes *val* visible only to the process executing the operation. It implements Local Consistency described in Section 2.2.2.

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 and with the information flow through weaker writes. 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. This implies that if a process $i$ reads a value written by process $j$ then process $i$ must be aware of all previous (in $\mathbf{R}_j$) writes by $j$ that are at least as strong as the write that is read. This condition does not hold slow *Slow_Write* as they may not be propagated to all processes.

### Liveness Requirements

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 now impose the condition that all *CO_Writes* and *PRAM_Writes* be *eventually* propagated[3] to all processes. 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.

### Performance Implications

Our specification requires a total order order on all *CO_Write* events. This means that a *CO_Write* can return only after its position in the total order has been decided. Since any process could be doing a write at any time, this decision would require some kind of global consensus.

*PRAM_Writes* have to be propagated eventually. This implies that messages used to do the propagation have to be reliable. But they can be buffered so that the cost of their propagation can be amortized over several writes. Moreover, this also allows for overlapping communication with computation.

*Slow_Writes* do not require the messages to be reliable. The implementation can propagate them on a best-effort basis giving no guarantees about their propagation. The implementation of *Slow_Write* does not have to suffer any overhead for making the messages reliable. *Slow_Writes* can also benefit from buffering just as *PRAM_Writes* can.

*Local_Writes* require no communication at all.

## 3.2.2 Formal Specification

In this section we extend the formalism described in Chapter 2 to describe the hybrid executions permitted by MERMERA. This extension takes into account the fact that the Coherent behavior is affected by the information that may flow through weaker writes. In other words, applying the correctness conditions of Chapter 2 to projections of the different types of writes in $\Delta$ is not sufficient.

The *write* operations now have a *type* associated with them and if $\alpha$ is a write operation then its type is referred to by $\alpha$.type. The type is a member of a totally ordered set $T = \{co, ca, p, s, lc\}$. The total order $\langle T, \leq \rangle$ is reflexive ant it imposes the order $co \leq ca \leq p \leq s \leq lc$. Where necessary, we show the type of a write by using a subscript. Thus the write operations are $w_{co}$, $w_{ca}$, $w_p$, $w_s$, $w_{lc}$ and they represent Coherent, Causal, PRAM, Slow and Locally Consistent writes respectively.

We modify the constructors $\mathbf{R}_{ww}$ (Definition 2.7), $\mathbf{R}_{w/w}$ (Definition 2.10) and $\mathbf{R}_{r/w}$ (Definition 2.11) to use the type information. Each of them takes an additional parameter, $type \in T$.

[3] We will not dwell on the exact mechanism of propagation in this chapter. It could be done using update messages or invalidate messages. We will discuss update based protocols in chapters 4 and 5.

**Definition 3.1** $\mathbf{R}_{ww}(\Theta, type)$ *is defined on* $\Theta^w$ *as*
$(\alpha = x.w_i(v_1))\mathbf{R}_{ww}(\Theta)(\beta = x.w_j(v_2))$ *iff* $\exists((\gamma = x.r_k{:}v_2) \in \Theta)$ *such that* $(\alpha \mathbf{R}^+_{\Theta^w} \gamma) \wedge (\beta \mathbf{R}_{wr} \gamma) \wedge (\alpha.type \leq type) \wedge (\beta.type \leq type)$.

Note that $\mathbf{R}^+_{\Theta^w}$ may contain pairs due to writes weaker than *type*.

**Definition 3.2** $\mathbf{R}_{w/w}(\Theta, type) = \{(\alpha = x.w(v_1), \beta = x.w(v_2)) | (\alpha, \beta \in \Theta) \wedge (\alpha.type \leq type) \wedge (\beta.type \leq type) \wedge \exists(\gamma = x.r{:}v_2 \in \Theta | \beta \mathbf{R}^+_\Theta \alpha \mathbf{R}^+_\Theta \gamma)\}$

**Definition 3.3** $\mathbf{R}_{r/w}(\Theta, type) = \{(\alpha = x.r{:}v_1, \beta = x.w(v_2)) | (\alpha \in \Theta, \beta \in \Theta^w) \wedge (\beta \leq type) \wedge \exists((\gamma = x.w(v_1)) \in \Theta^w \wedge (\delta = x.r{:}v_2) | (\gamma.type \leq type) \wedge (\beta (\mathbf{R}^w_\Theta)^+ \alpha \mathbf{R}^+_\Theta \delta))\}$

**Definition 3.4** *A computation* $E = \langle \Delta, \mathbf{R} \rangle$ *is* Coherent (CO) *iff*
$\langle \Delta, (\mathbf{R} \cup \mathbf{R}_{ww}(\Delta, co))^+ \rangle$ *is a partial order.*

**Definition 3.5** $\mathbf{R}_{Causal} = \mathbf{R}^+ \cup \mathbf{R}_{r/w}(\Delta, ca) \cup \mathbf{R}_{w/w}(\Delta, ca)$. *A computation is* Causal *iff* $\langle \Delta, \mathbf{R}_{Causal} \rangle$ *is a partial order.*

**Definition 3.6** $\mathbf{R}_{PRAM}(\Delta_{ij}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta, p) \cup \mathbf{R}_{r/w}(\Delta_i, p)$, *where* $\Theta = \Delta_{ij}$. *A computation is a* PRAM *computation iff* $\forall i, j \in P$, *the set of all processes,*
$\langle \Delta^w_{ij}, \mathbf{R}_{PRAM}(\Delta_{ij}) \rangle$ *is a partial order.*

**Definition 3.7** $\mathbf{R}_{Slow}(\Delta_{ijx}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta, s) \cup \mathbf{R}_{r/w}(\Delta_{ix}, s)$, *where* $\Theta = \Delta_{ijx}$. *A computation is a* Slow *computation iff* $(\forall i, j \in P, \forall x \in L$, $\langle \Delta^w_{ijx}, \mathbf{R}_{Slow}(\Delta_{ijx}) \rangle$ *is a partial order.*

**Definition 3.8** $\mathbf{R}_{LC}(\Delta_{ix}) = (\mathbf{R}^w_\Theta)^+ \cup \mathbf{R}_{w/w}(\Theta^w, lc) \cup \mathbf{R}_{r/w}(\Theta, lc))$, *where* $\Theta = \Delta_{ix}$. *A computation is* Locally Consistent *iff* $\forall i, x$, $\langle \Delta^w_{ix}, \mathbf{R}_{LC}(\Delta_{ix}) \rangle$ *is a partial order.*

A hybrid execution is correct iff it is Coherent, Causal, PRAM, Slow and locally consistent. Note again that this model does not have any liveness requirement.

## 3.3 Using Mermera

Having specified the behavior of Mermera we will now give two examples showing how it can be used. We use all the behaviors of Mermera to solve a system of linear equations using an asynchronous iterative method [BT89][4]. We also show how the operations of Mermera can be used to achieve barrier synchronization.

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

```
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);
    }
  }while (AbsoluteDiff < Epsilon)                         /* Local termination check */
  for (i = MyLow; i < MyHigh; i++)
    PRAM_Write(XStartLoc + i, Read(XStartLoc + i));     /* Ensure that values
                                                           propagate to all processes
                                                           */
  barrier();                      /* Wait for all processes to satisfy local termination */
}while (!global_termination());
```

Figure 3.2: A linear equation solver.

### 3.3.1 Solving a System of Equations

In this section we present a program that implements an asynchronous iterative algorithm to solve a linear system of equations $Ax + b = 0$, using non-coherent memory on $p$ processes. $x$ and $b$ are vectors of size $n$ and $A$ is an $n \times n$ matrix. As mentioned earlier the asynchronous iterative method to solve this system will converge if $\rho(|A|) < 1$.

The program shown in Figure 3.2 is executed by each of the $p$ participating processes. Each process except the $p^{th}$ process computes $\lfloor \frac{n}{p} \rfloor$ elements of $x$. Process $p$ computes $n - (\lfloor \frac{n}{p} \rfloor) \times (p-1)$ elements. 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 values of $x$ produced in the last iteration before the local termination are propagated to all processes. Then each process does a global termination check by running an iteration to compute *all* elements of $x$. If the check succeeds, the program terminates with process 1 writing the solution to a file.

Our specification of Slow memory permits MERMERA to propagate *Slow_Write*s on a best-effort basis, *i.e.*, the system does not guarantee propagation of values written using *Slow_Write* to all processes. It is for this reason that every process uses *PRAM_Write* after every local termination. This satisfies the progress requirement mentioned in Section 2.3. In this application, a process does $n-1$ *Read* operations on shared memory for computing each $x_i$ that it is supposed to compute. It then does one *Slow_Write*. Therefore it does $n-1$ *Reads* for each write operation. An implementation of MERMERA that makes *Read* operations cheap would be ideal for this application.

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

Figure 3.3: Pseudo C code for process $i$ to implement barrier synchronization. Locations 1 through (NumOfProcs + 1) in shared memory are reserved for use in barrier synchronization.

### 3.3.2 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 3.3) 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.

We cannot use *Slow_Write* 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.

In the next two chapters we describe the implementation of MERMERA on a BBN Butterfly TC2000 and on a network of SUN SPARCstation 1+ workstations. We will also describe the performance of the linear solver described in Section 3.3.1.

# Chapter 4

# A Pilot Study on a BBN Butterfly

In the last chapter we established that Non-coherent memories could be used for certain applications. We also argued that the implementation of Non-coherent memories would be more efficient than that of Coherent memory.

In this chapter we describe a pilot implementation on a BBN Butterfly TC2000. The purpose of this implementation is to get a feel for the the improvement in access time that one can get by making the memory non-coherent. Section 4.1 describes the algorithms to implement a subset of the operations provided by MERMERA. We implement *CO_Write*, *PRAM_Write* and *Read* operations. Our algorithms favors *Read* operations by using a fully replicated update based protocol. In Section 4.2 we present performance measurements from our implementation.

## 4.1 Implementation

Having described the behavior of MERMERA, we now present a description of a pilot implementation on a BBN Butterfly TC2000 using the Uniform System [Mat90]. We implement *PRAM_Write*, *CO_Write* and *Read* operations. The purpose of this partial implementation, which does not support hybrid computations, is to get a feel for the improvement in memory response time that one may get by tolerating weaker coherence.

The TC2000 is a distributed memory machine, where a remote memory reference takes about 2.1 microseconds—three times longer than a local memory reference. The particular machine used has 45 processors, which run in a dedicated mode, *i.e.*, one process per processor.

We implement update based protocols for *CO_Write* and *PRAM_Write*. Each process has a copy of the shared locations. All writes are broadcast to all processes and reads return the value in the local copy of the location being read.

We have two implementations of *CO_Write*. The first uses traditional two-phase locking (2PL), and the second employs a *pipelined locking* protocol (PLP) that is much more efficient than 2PL under certain loads. These protocols are given in sections 4.1.1 and 4.1.2. Section 4.1.3 describes an algorithm to implement PRAM. All these algorithms are programmed as macros to avoid the penalty of an additional function call for each read or write operation. We find that the PRAM outperforms both the 2PL and PLP implementations of coherent memory, by at least an order of magnitude, even under loads that maximize

PLP's ability to pipeline memory requests efficiently.

### 4.1.1 Two phase locking (2PL)

We use a traditional 2PL protocol, in which a write obtains locks on all copies of a memory location before it releases any of them, and a read operation locks and reads only the local copy of the memory location. Read and write locks are not distinguished, since concurrent reads never attempt to obtain the same lock. Thus we ensure the serializability of all coherent read and write operations. Sequential consistency is guaranteed by additionally respecting program ordering, by blocking a process that invokes a memory operation until that operation is completed. To avoid deadlocks, we require all processes to obtain the locks in the same fixed order, a technique often termed *conservative* 2PL.

### 4.1.2 Pipelined locking protocol (PLP)

In this protocol we associate a single lock with the entire copy of the shared memory at each process. When a copy of any shared location in a process has to be modified, the whole copy of shared memory at that process is locked. Writes are propagated to copies in a fixed order, with the lock being acquired before the value is written, and released only after the next copy's lock is acquired. The advantages of this protocol are that shared memory reads do not require any locking, and that write locks are held only for the duration of updating one copy of the memory location. Thus, multiple writes *to the same location* can proceed concurrently in a pipelined fashion, as long as they do not require access to the same copy of memory at the same time. This makes PLP suitable for shared memories with many hot spots. Its disadvantage is that concurrent writes to different locations are also pipelined.

The PLP algorithm is given in Figure 4.2, where *UsLock* and *UsUnlock* are atomic operations provided by the Uniform System on BBN Butterfly. *MemCopyPtrs[i]* contains the lock for the copy of shared memory at process $i$ and a pointer to the copy of the $0^{th}$ location at process $i$. A *read* operation does not need any locking—it simply returns the value in the local copy of shared memory.

### 4.1.3 PRAM algorithm

The algorithm for *PRAM_Write*, described in Figure 4.3, does not require any locking. The condition that a process' writes be observed in the order they are done at the writer is trivially satisfied because the writer blocks until the value is written in all processes' copies of the shared memory.

## 4.2 Performance Results

In this section we report our performance measurements of this pilot implementation.

### 4.2.1 Access Time

We measure the performance of *PRAM_Write*, the two implementations of *CO_Write* (*i.e.*, *2PL_Write* and *PLP_Write*) and the corresponding Read operations. We give the average

```
#define TIMEOUT 10                                      /* Number of microseconds to wait before retrying for a lock */

struct LocCopyStruct
{ short Lock;                                           /* Per copy per location */

  int Value;};

struct LocCopyStruct *MemCopyPtrs[MAXPROCS];
int NumProcs;

CO_Write_2PL(int Loc, Val)                              /* Requires the use of Read_2PL */

{ register int i;
  register struct LocCopyStruct *LocCopyPtr;

  for(i = 0; i < NumProcs; i++)
  { LocCopyPtr = MemCopyPtrs[i] + Loc;
    UsLock(&(LocCopyPtr -> Lock), TIMEOUT);
    LocCopyPtr -> Value = Val;}
  for(i =0; i < NumProcs; i++)
    UsUnlock(&((MemCopyPtrs[i] + Loc)->Lock));
}

Read_2PL(int Loc, *Buffer)                              /* Must use instead of Read with Co_Write_2PL */

{ register struct LocCopyStruct *LocCopyPtr;

  LocCopyPtr = MemCopyPtrs[UsProc_Node] + Loc;
  UsLock(&(LocCopyPtr -> Lock), TIMEOUT);
  *Buffer = LocCopyPtr -> Value;
  UsUnlock(&(LocCopyPtr -> Lock));
}
```

Figure 4.1: The two phase locking protocol.

```
#define TimeOut 10                      /* Microseconds to wait before retrying for a
                                           lock */
struct MemCopyPtrStruct
{ short *LockPtr;                       /* A lock per memory copy */
  int *LocPtr;};                        /* Pointer to copy of 0^th variable in shared
                                           memory */
struct MemCopyPtrStruct MemCopyPtrs[MaxProcs];

CO_Write_PLP(int Loc, Val)
{register int i=0, j, k;

  UsLock(MemCopyPtrs[i].LockPtr, TimeOut)
  for(i = 1; i < NumProcs; i++)
  { *(MemCopyPtrs[i-1].LocPtr + Loc) = Val;
    UsLock(MemCopyPtrs[i].LockPtr, TimeOut);
    UsUnlock(MemCopyPtrs[i-1].LockPtr);
  }
  *(MemCopyPtrs[i-1].LocPtr +Loc) = Val;
  UsUnlock(MemCopyPtrs[i-1].LockPtr);
}

Read(int Loc, *Buffer)
{ *Buffer = *(MemCopyPtrs[UsProc_Node].LocPtr + Loc);
}
```

Figure 4.2: The Pipelined Locking Protocol.

```
struct MemCopyPtrStruct
{ int *LocPtr; }                        /* Pointer to copy 0^th location in shared
                                           memory */
struct MemCopyPtrStruct MemCopyPtrs[MaxProcs];

PRAM_Write(int Loc, Val)                /* Use Read from PLP. /*
{ register int i;
  for(i = 0; i < NumProcs; i++)
  *(MemCopyPtrs[i].LocPtr + Loc) = Val;
}
```

Figure 4.3: The PRAM algorithm.

of the response times of all processes for a certain number of *read/write* operations to a single location.[1]

### Experimental Methodology

Each process does the same operations on the location in shared memory. The total response time seen by each process for a given load is added together and the sum is divided by the number of processes to obtain the average response time for that load.

We report on measurements for three different sets of loads. The first set has 1000 Reads and zero Writes, the second set has 500 Reads and 500 Writes, while the third set has zero Reads and 1000 Writes. We vary the number of processes from 1 to 30, where each process runs on a separate processor and each process imposes the load above. Every measurement was repeated at least ten times, with negligible variance in the results.

Plots from our measurements are shown in figures 4.4, 4.5, and 4.6. In Figure 4.4 we show the performance of reads that require locking the local copy (*i.e.*, *Read_2PL*) and of reads that do not (*i.e.*, *Read* defined in Figure 4.2). This gives us an estimate of the cost of obtaining a local lock and releasing it (about 2.25 microseconds). Consequently, reads can be considered of zero cost relative to writes.

Figures 4.5 and 4.6 show the plot of the average response time against the number of processes for two sets of loads. These show the performance of the system under a load of only *Write* operations and under a load in which 50% of the operations are *Read* operations and 50% are *Write*s.

### Discussion of Measurements

In all three methods the cost of writing dominates the cost of doing the corresponding read. Our most important observation is that *PRAM_Write* is faster[2] than *CO_Write* by a factor of about 40 in the case of the two phase locking implementation and by a factor of about 10 in the pipelined locking implementation. These measurements indicate that non-coherent memory possesses at least a ten-fold advantage over coherent memory in terms of response time. This performance advantage is robust against significant optimization of Coherent memory implementation, and against a low latency differential between remote and local communication. However, this implementation does not exploit the fact that *PRAM_Write* can be buffered and computation and communication can be overlapped.

## 4.2.2 Solving a System of Equations

In this section we briefly present the performance of the equation solver described in Section 3.3.1.

[1]This choice of load, while simplistic, favors coherent memory because it enables PLP to exhibit its best performance.

[2]At 30 processors (processes), 500 PRAM Reads and 500 PRAM Writes took 48 milliseconds to finish, while 1000 writes took about 94 milliseconds to conclude.

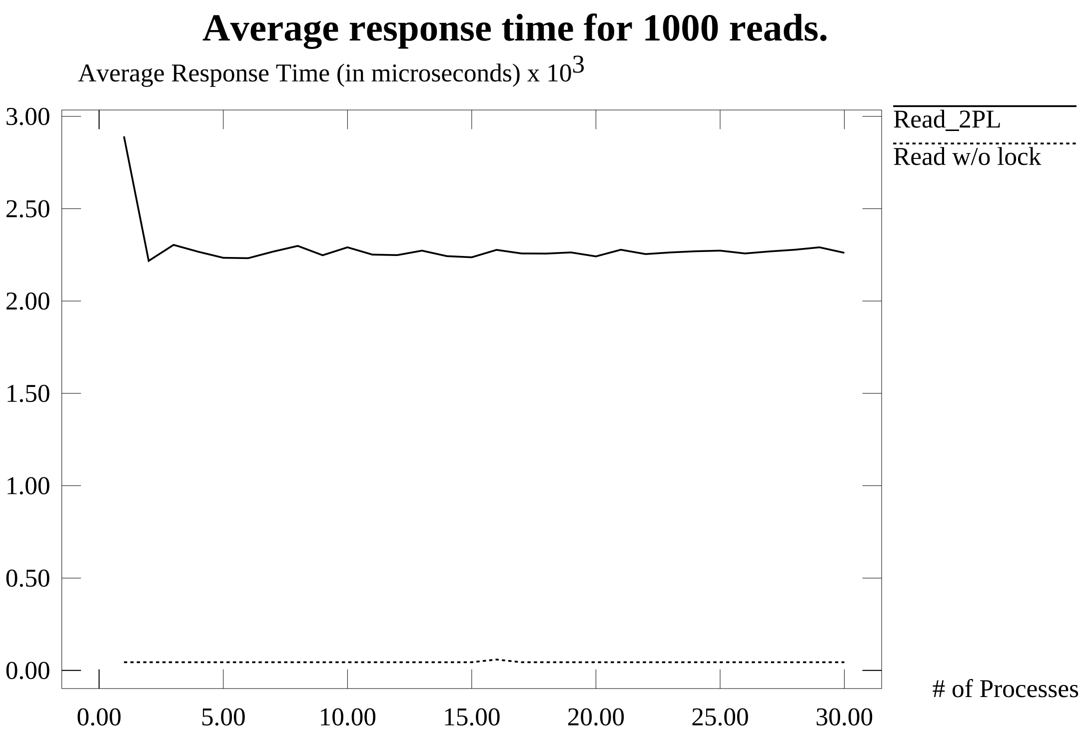


Figure 4.4: Performance of Read operations. All times are in microseconds.

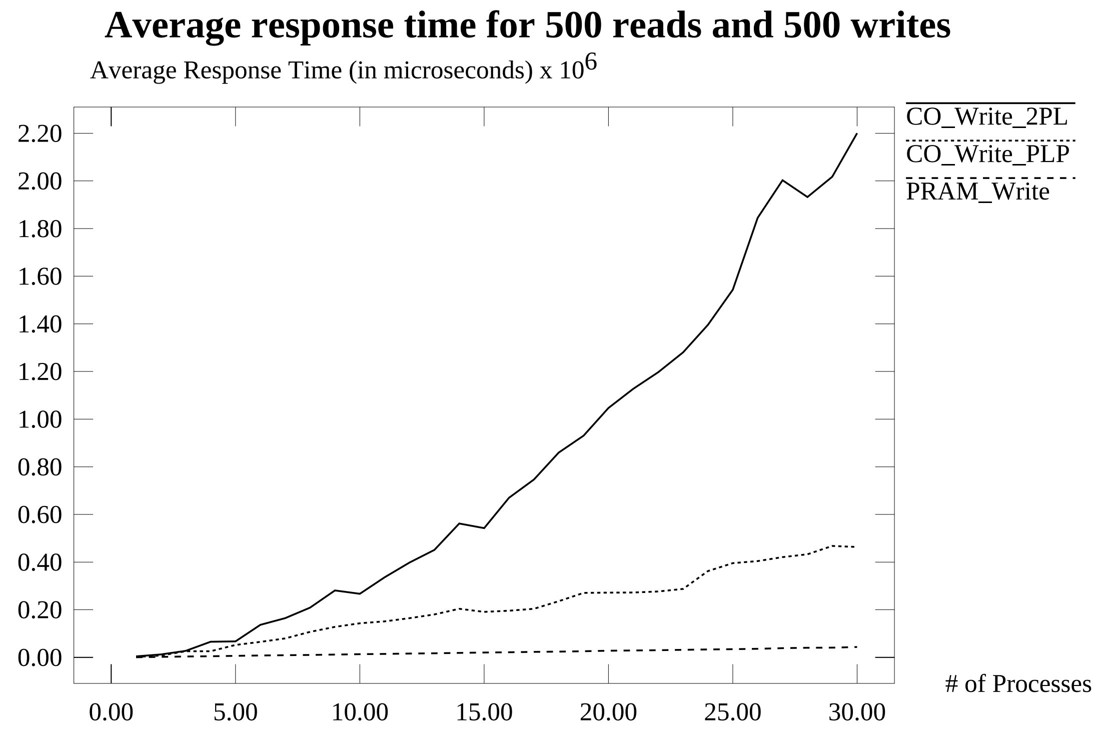


Figure 4.5: Performance of Read and Write operations. All times are in microseconds.

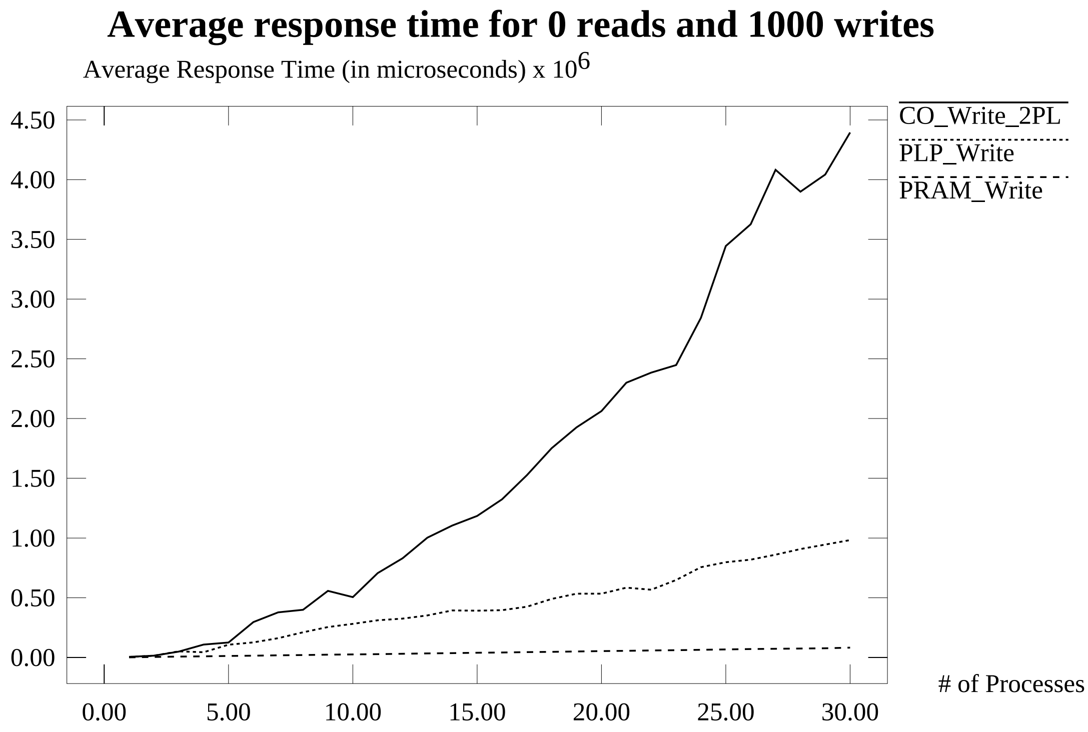


Figure 4.6: Performance of Write operations. All times are in microseconds.

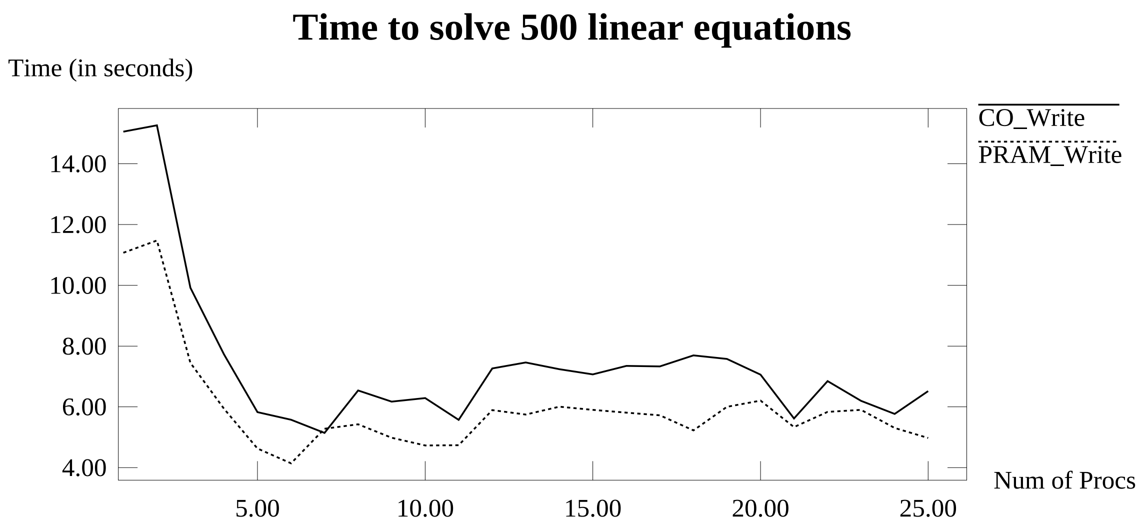


Figure 4.7: Performance of the solver on the TC2000.

### Experimental Methodology

The program shown in Figure 3.2 was implemented. The *Slow_Writes* were replaced by *PRAM_Writes*. This does not affect the correctness of the algorithm because computations allowed by PRAM are also allowed by Slow Memory. The performance of this program was compared with a program that used *CO_Writes* instead of *PRAM_Writes*. Between 1 and 25 processors were used to solve a system of equations in 500 variables and the time taken for the iteration to converge was measured. The time taken for the convergence to be detected was included in the measurements. Our measurements are summarized in Figure 4.7.

### Discussion of Measurements

The program using *PRAM_Writes* performs up to 30% faster than the program using *CO_Writes*. The poor improvement can be attributed to two factors. First, the difference in the two programs is in the way they do their writes to the shared-memory. But the number of writes is a very small fraction[3] of the operations done in the iterations. Therefore, optimizing them even by an order of magnitude improves the overall performance by only a small amount. Second, *PRAM_Writes* allow writes to be buffered and the communication costs can be amortized over several writes. But our implementation does not exploit this. An implementation that takes advantage of this freedom will be able to amortize the cost of remote communication over several writes.

The performance of the solver is rather discouraging. It warns us that in the absence of buffering achieving even a 10-fold improvement in response time may not be sufficient to get a significant improvement in the performance of certain applications. The read/write mix of the application also plays a significant role in the performance. In the next chapter we

[3]For each *write* to shared memory each process does $2n$ reads of local memory, $n$ additions and 1 division, where $n$ is the number of variables in the system of equations being solved.

present an implementation of MERMERA on a network of SUN SPARCstation 1+ workstations. In that implementation we buffer writes and we achieve a much higher performance improvement by using non-coherent memory.

# Chapter 5

# A Prototype on a Network of Workstations

In this chapter we describe an implementation of MERMERA on a network of SUN SPARC-station 1+'s using the Isis toolkit. This implementation takes advantage of the fact that non-coherent writes to the shared memory can be buffered. The performance of our implementation is described in Section 5.2. Our experiment demonstrates the performance improvement that can be achieved by certain applications by using non-coherent memory. In Section 5.3 we discuss the impact of Isis on our implementation.

## 5.1 Isis Implementation

In this section we describe an implementation of MERMERA on a network of SPARCstations running Unix. We use version 2.2.5 of the Isis toolkit [BSS91, BJ87] for our implementation. Our algorithms are update based, *i.e.*, each process has a copy of the shared memory and this copy is updated as writes occur.[1] A *Read* returns the value in the local copy. This is in concurrence with our decision to make *Read* operations cheap. 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.

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.[2] The only guarantee is that the correctness conditions will not be violated. According to the specification of MERMERA, *Local_Writes* are not propagated. Only the local copy is updated. The weaker ordering constraints on non-coherent operations also allow us to buffer multiple writes and propagate them together.

The writes are propagated by multicasting the values to other processes. We use the Isis toolkit for our implementation because it gives us a suite of multicasts to groups of processes which satisfy different ordering properties. The multicasts of interest to us are

[1] The implication of this full replication on the performance comparisons is discussed in Section 5.2.

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

*abcast(), fbcast()* and *mbcast()*. These primitives have different constraints on the order in which the messages are delivered to their destinations. We summarize these differences in the next few paragraphs.

*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 by 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.[3] 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 implementation and ensure that the correctness conditions of MERMERA are satisfied. Our technique 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[4]. These algorithms are given in figures 5.2, 5.3, 5.4 and 5.5. All writes other than *CO_Writes* may be buffered. The structure of the buffer is shown in Figure5.1. Its size can be dynamically adjusted[5]. 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 Figure2.11 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

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

[4]In Section 5.3 we mention some optimizations that can be made to our implementation.

[5]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 5.1: Structure of buffer. This buffer is the message that is broadcast.

effort policy does not require this reliability. As discussed in Section 5.3 we may switch to an unreliable broadcast protocol for propagating *Slow_Writes*.

When an update message is delivered to a process a task specified by the *update_memory* function (Figure5.4) 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[6] 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.

## 5.2 Performance Results

In this section we describe the performance of our implementation. We conduct two types of experiments. First, we measure *access time* and *completion time* (defined in the following subsection). Second, we measure the performance of the equation solver described in Section 3.3.1 on our implementation. Two versions of the solver are used. The first version uses all the behaviors of MERMERA while the second version uses coherent behavior only.

### 5.2.1 Access Time and Completion Time

The *access time* of an operation is defined to be the duration of time between the invocation and return of the operation. In case of non-coherent write operations this does not imply that when the write returns, the value written has been propagated to all processes sharing

[6]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 5.2: 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 5.3: Pseudo C code for *broadcast*().

```
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.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;
        CurrentBuffer = enabled_Buffer(Sender);}
}
```

Figure 5.4: Pseudo C Code for the handler for the *update_memory* message. The function *enabled_Buffer*(Sender) is described in Figure5.5. 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 5.5: Pseudo C code for *enabled_Buffer()* function.

memory. To measure the time taken for the operation 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 the propagation of all these ($p \times 100$) writes to be propagated to all the $p$ participating processes.

### Experimental Methodology

Our experiments are conducted on a dedicated network of 6 SPARCstations. 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). 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 the different processes may be different. This is especially true for *CO_Writes* which relies on the total order of all *abcasts*. Isis achieves this total order by designating one of the participating processes, $p_0$ to be the sequencer. A consequence of this is that *CO_Writes* by process $p_0$ have a much faster *access time* than other processes because they do not need

| Procs | CO_Write | PRAM_Write | | | | Slow_Write | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Buffer Sizes | | | | Buffer Sizes | | | |
| | | 1 | 10 | 100 | 1000 | 1 | 10 | 100 | 1000 |
| 1 | 133.0 | 46.9 | 5.9 | 1.2 | .66 | 46.9 | 5.5 | 1.2 | .62 |
| 2 | 532.0 | 108.0 | 16.8 | 2.5 | .66 | 96.4 | 16.1 | 2.4 | .62 |
| 3 | 1005.0 | 101.3 | 26.6 | 3.9 | .66 | 80.6 | 25.9 | 3.9 | .62 |
| 4 | 2049.0 | 110.8 | 28.1 | 5.2 | .66 | 86.8 | 27.8 | 5.1 | .62 |
| 5 | 3561.0 | 117.8 | 27.7 | 6.3 | .67 | 86.4 | 27.2 | 6.3 | .63 |
| 6 | 5378.0 | 120.5 | 28.0 | 7.1 | .66 | 91.4 | 27.1 | 7.1 | .62 |

Table 5.1: Access time in milliseconds for 100 writes. Buffer Size is measured in terms of the number of write operations that can be buffered.

any remote communication to determine their position in the total order. The average over the access time observed by each process is reported.

To measure the completion time, each process performs 100 writes between two barrier synchronization calls. Again, the average of the times observed by each process are reported. Tables 5.1 and 5.2 list our measurements.

### Discussion of Measurements

For a buffer size of 1000, 100 *PRAM_Writes* take 0.66 milliseconds which shows that it takes $6.6\mu s$ to execute the code that buffers each write. The code to buffer each *Slow_Write* takes $6.2\mu s$.

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 access time and the information flow time. This is because the buffer size determines the number of messages that are sent and received. The access 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 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 the writing process before it can continue. The message will be delivered 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 a delay in the resumption of the waiting task. This explains the faster than linear (in the number of processes) growth of the access time of *CO_Writes*.

The completion 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 completion time. This number depends on the buffer size. However, if we do at most 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 completion time. For small buffer sizes the writes generate a large number of small messages. Isis can coalesce these small messages to larger messages for more efficient transmission. This explains the observation that for a given number of

| Procs | CO_Write | PRAM_Write | | | Slow_Write | | |
|---|---|---|---|---|---|---|---|
| | | Buffer Sizes | | | Buffer Sizes | | |
| | | 1 | 10 | 100 | 1 | 10 | 100 |
| 1 | 136.8 | 50.4 | 9.4 | 4.8 | 46.8 | 9.0 | 4.7 |
| 2 | 1030.0 | 267.2 | 54.2 | 26.2 | 245.3 | 53.9 | 27.1 |
| 3 | 1574.0 | 412.9 | 100.0 | 52.4 | 383.4 | 99.7 | 53.9 |
| 4 | 2889.0 | 638.0 | 148.0 | 85.8 | 601.4 | 145.8 | 85.7 |
| 5 | 4645.0 | 963.8 | 188.4 | 124.9 | 871.9 | 187.9 | 125.4 |
| 6 | 6761.0 | 1204.0 | 247.9 | 170.9 | 1223.0 | 246.8 | 174.0 |

Table 5.2: Completion time in milliseconds for 100 writes.

processors the completion time grows at a rate *sub-linear* in the number of messages sent.

The access time and completion 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 its 100 writes it would have already processed many of the writes by other processes.

Table 5.2 shows that for a fixed buffer size the completion time of non-coherent memories depends linearly on the number of processes sharing memory. On the other hand the completion time of coherent writes grows faster than linearly as the number of processes is increased. Completion times for *CO_Writes* is 3-38 times slower than the non-coherent writes. The factor increases as the number of processes increases.

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 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 3.3.1) 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 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 the implementation described.

### 5.2.2 Solving a System of Equations

In this section we study the performance of the linear equation solver described in Chapter3. The performance of the solver that uses the non-coherent behaviors of MERMERA is compared with a program that uses the coherent behavior only. The effect of buffer size on the performance of the solver is discussed.

```
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
                                                           propagate to all processes
                                                           */
   barrier();                  /* Wait for all processes to satisfy local termination */
}while (!global_termination());
```

Figure 5.6: Solver with *isis_accept_events()*.

### Experimental Methodology

In Chapter3 we presented a program to solve a linear system of equations, $Ax + b = 0$, using the different operations offered by MERMERA. The program is executed with a minor modification shown in Figure5.6. The modification is the addition of a call to the function *isis_accept_events* which allows Isis to deliver messages received from other processes.

The program solves a system of randomly generated[7] 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 the convergence to be detected. This time is called the *convergence time* 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 Figure5.6 is compared with 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 was different and it varied from run to run.

[7]To ensure $\rho(|A|) < 0$, all elements of $|A|$ are less than 1 and to ensure the numerical stability of the algorithm the diagonal elements of $A$ are much larger than the other elements.

| Procs | $T_{CO}$ | $T_{mixed}$ | $\frac{T_{CO}}{T_{mixed}}$ |
|---|---|---|---|
| 1 | 158.2 | 157.1 | 1.01 |
| 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 5.3: Performance measurements for $n = 1000$. All times in seconds.
$T_{CO}$ = time taken for iteration to converge using coherent writes only.
$T_{mixed}$ = time taken for convergence using mixed behavior shown in Figure5.6.
A purely sequential Gauss-Seidel iteration that does not use MERMERA converged in 143 seconds after doing 42 iterations.

### Discussion of Measurements

Our measurements are summarized in table 5.3 and plotted in Figure5.7. The fastest convergence time regardless of the buffer size are used in these figures.

The main conclusions we derive from these measurements are:

1. Using 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 program using coherent memory does not give any performance improvement when we use more than one processor. Using multiple processes always takes longer than using a sequential Gauss-Seidel iteration. On the other hand, *using non-coherent behavior gives us better performance.*

3. The performance improvement the program using coherent behavior breaks down at 5 processors while the performance of the program using non-coherent behavior continues to improve up to at least 6 processors.[8]

**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 Figure5.8 show that the buffersize has a significant effect on the performance of the solver.

For any number of processors the performance is the best when the buffersize 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

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

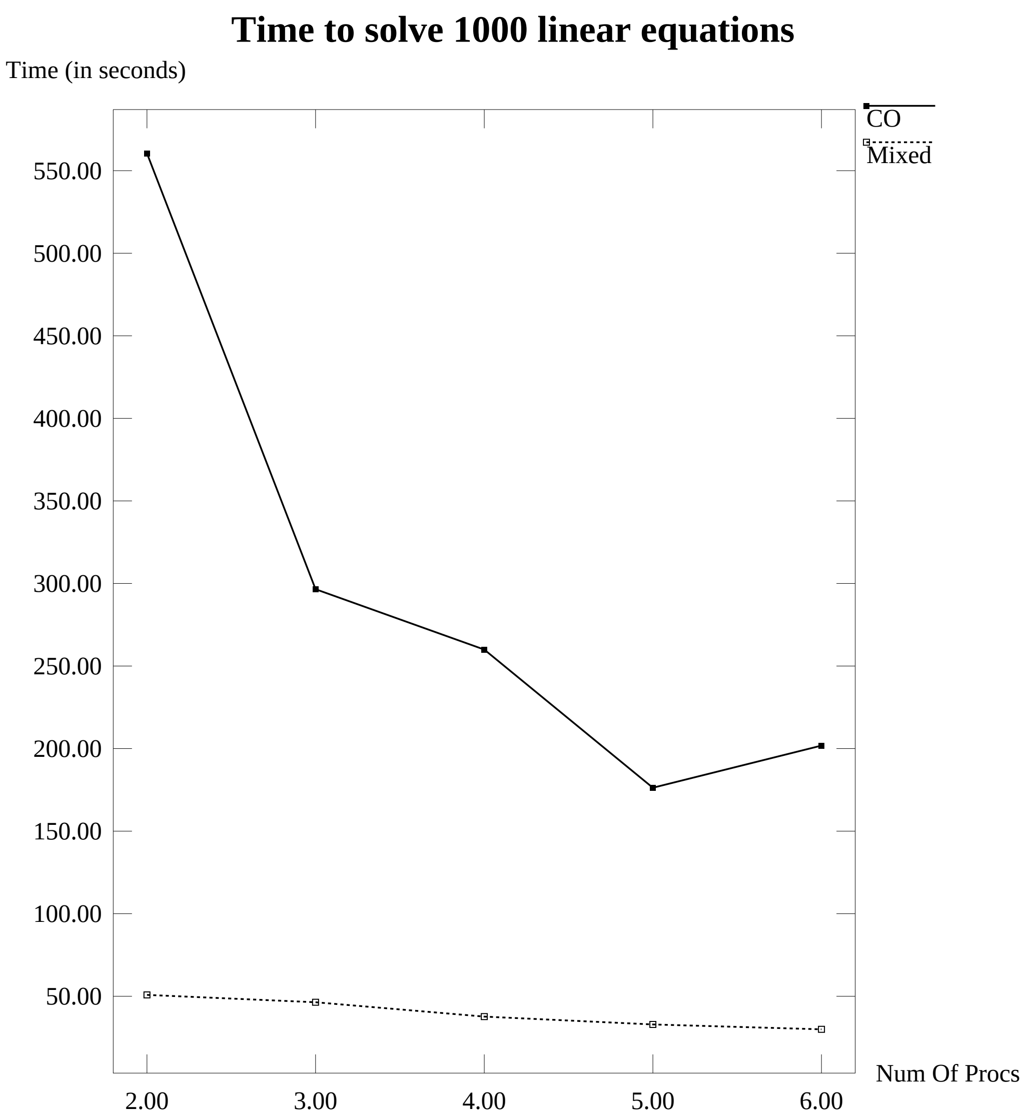


Figure 5.7: Convergence time vs. Number of processors.

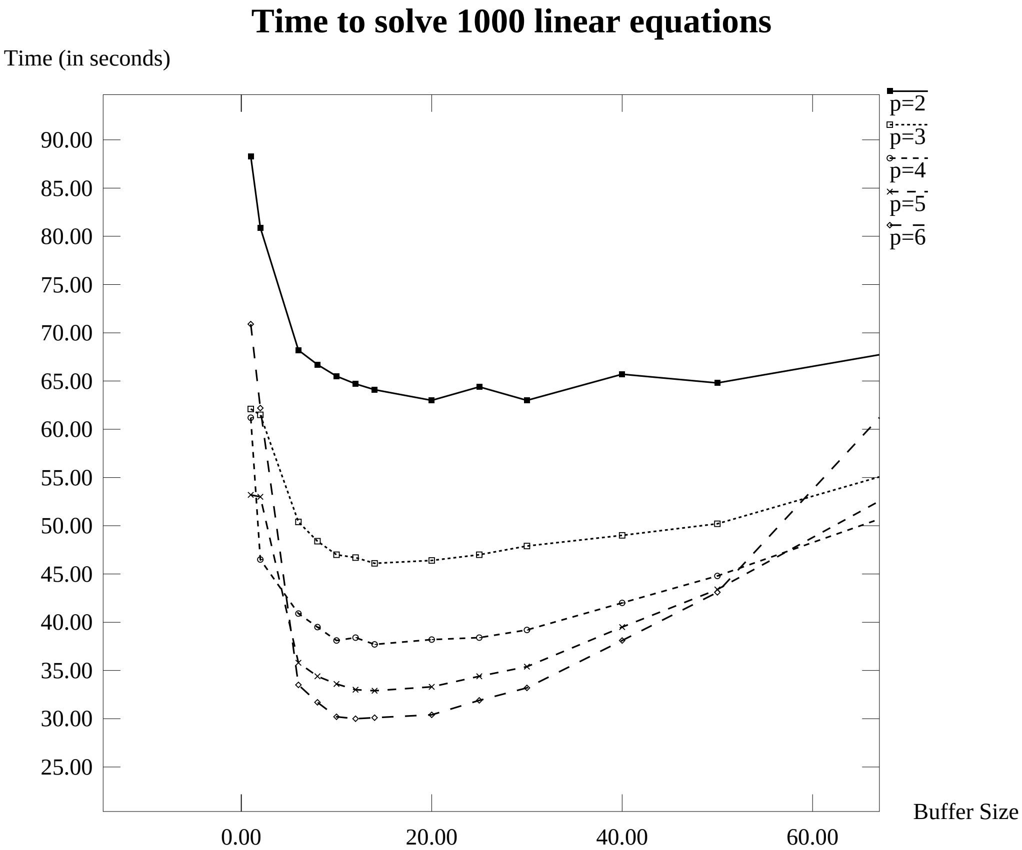


Figure 5.8: Effect of Buffersize on performance.

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 buffersize is 500.[9]

## 5.3 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 parts of the code external to Isis is very small. The implementation of Isis is highly optimized for the functionality it offers.

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

[9]For presentation reasons we have excluded that data point from Figure5.8.

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.

In this chapter we described the implementation and performance of MERMERA on a network of SPARCstations. The performance of a solver that uses the different behaviors of MERMERA was compared with that of one that used only its coherent behavior. This comparison corroborates our belief that non-coherent memory can be a performance boon for some applications.

# Chapter 6

# Conclusion

In Section 6.1 of this chapter we summarize the conclusions of our study and in Section 6.2 we speculate on the impact of technological advances on our results and explore directions for future research.

## 6.1 Summary

In this thesis we presented MERMERA, a model for parallel computing using non-coherent shared memory (Figure 6.1). It offers programmers an opportunity to trade off programming simplicity for performance. Programming with our system is more complex than using *Read* and *Write* operations of coherent shared memory because of the existence of different types of *Write* operations, but, as shown in our example of the linear solver, significant improvements in performance can be achieved over using coherent shared memory. In Chapter 2 we developed a formal model to describe the non-coherent behaviors proposed in the literature. We used this model to prove that totally asynchronous iterative algorithms converge on Slow Memory — a non-coherent memory proposed in [HA90]. As an example of such an algorithm, we presented a program to solve a linear system of equations. We extended our model in Chapter 3 to describe the hybrid behavior of MERMERA.

Chapters 4 and 5 described the the implementation and performance of our model

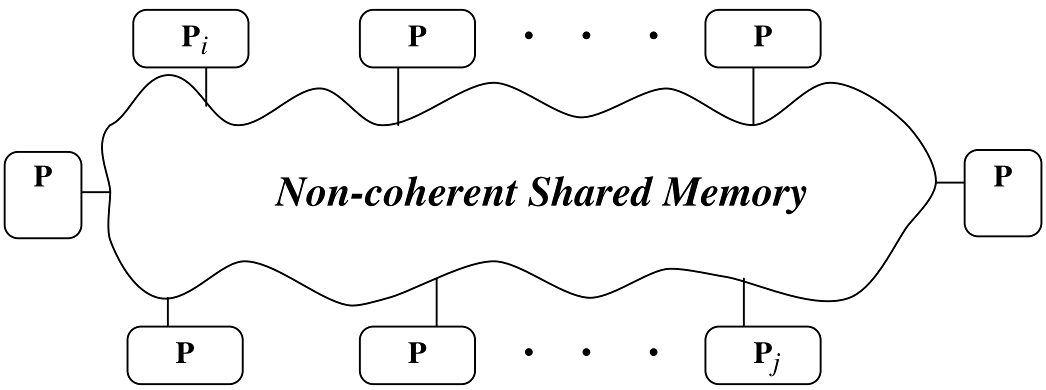


Figure 6.1: Non-coherent shared memory model.

on a BBN Butterfly TC2000 and on a network of SPARCstation 1+'s. We compared the performance of the different types of writes of MERMERA using the access time and completion time measures. We also compared the performance of a version of the linear solver that uses non-coherent operations of our model with that of a version that uses only the coherent behavior. In our experiments on a network of SPARCstations, the former outperformed the latter by a factor ranging from 5 to 11. These results were presented in Section 5.2.2.

## 6.2 Future Work

This thesis showed how a totally asynchronous algorithm can achieve a significant improvement in performance by using a hierarchy of non-coherent behaviors. Totally asynchronous iterative algorithms for several applications have been proposed in chapter 6 of [BT89]. A technique to easily convert these algorithms for MERMERA would be very useful. In general, we expect programmers to use coherent writes for the first implementation of an algorithm. They can then identify the portions of the program that can tolerate non-coherence and get an improved performance.

Of still greater importance is to find other applications than can benefit in performance by employing non-coherent memory. One possible area for exploration is the use of non-coherent memory in real-time systems. Real-time systems have the characteristic that computations have deadlines. Sometimes, an imprecise result of a computation may be preferred to a missed deadline. A notion of imprecise computations has been proposed in [CLL90]. We know that non-coherent memory operations complete faster than coherent operations but they offer weaker ordering guarantees. A computation may choose to use non-coherent operations and provide imprecise results rather than use coherent operations and miss its deadline.

The architecture community has attacked the problem of poor performance of Coherent shared memory by offering weaker consistency constraints,*e.g.*, *processor consistency* [Goo91], *weak ordering* [AH90], *release consistency* [LLG$^+$90] and *entry consistency* [BZ91]. All of these, except *processor consistency*[1], require the programmer to distinguish between synchronization accesses and ordinary accesses to memory. If the programmer makes this distinction then the programs written for sequentially consistent memory will execute correctly on these memories. Compiler techniques to automate this process are being studied. Extending our model to incorporate this distinction between synchronization accesses and ordinary accesses is another research issue. In the same vein, it would be interesting to extend the formal model to allow shared objects with arbitrary operations (*e.g.*, incr/decr, enque/deque).

In Chapter 5 we used *access time* and *completion time* as metrics to measure the performance of our implementation. In the same chapter we presented the performance of a linear equation solver. In terms of the metrics, the system performed the best for the largest buffer size. On the other hand, our application performed the best when the buffer size was 12-14. We could not derive any correlation between the performance of our system on the

[1]Processor Consistency is similar to PRAM. In addition to the correctness condition of PRAM, it requires that there be a total order on all writes to a location and each process' view of the memory be consistent with this order.

metrics above and the performance of our application. Research in the area of developing performance metrics for non-coherent memories that can help us predict the performance of applications on these memories can be very useful. Such a metric would allow a memory designer to design memories that optimize these measures without having to be concerned with individual applications. However, we are not sure about the *existence* of such a metric. In the absence of such a metric, the programmer should have some control on the parameters that affect the performance of the memory. Compiler techniques to adaptively set these parameters can also be developed.

Our implementation uses full replication of the shared memory. Alternative implementations can use partial replication. One such protocol using a directory-based scheme for *PRAM_Writes* is described in [LS88]. Enhancing such a protocol for a hierarchy of memory behaviors is another area that deserves attention. But, first we need to identify applications that can benefit from such an implementation. Our example of the linear solver benefits from full replication because the data written by any process is used by all other processes.

Advances in networking technology [ABC+89] have introduced the idea of having a network co-processor at each workstation. This co-processor takes over the burden of sending and receiving messages. Applications using non-coherent writes can benefit from such an enhancement because then there can be true parallelism in the overlapping of communication and computation. While a message (generated by a non-coherent write) is being multicast by the co-processor the CPU can proceed with the computations of the thread doing the non-coherent write. Applications using coherent writes cannot benefit as much from this enhancement because a thread doing a coherent write can proceed only after the *CO_Write's* position in the total order of *CO_Writes* is determined. In general, the time taken for this will depend on the number of processes in the system.

In the future, we can expect the latency of networks that connect distributed systems to decrease and their bandwidth to increase. But the decreases in latency will be small relative to the increases in bandwidth. This is because a large part of the latency consists of instructions in the operating system code at the nodes. How will this affect the performance of non-coherent memory versus that of coherent memory? Will the difference disappear? Our conjecture is that, as long as the time to access local memory is 3-4 orders of magnitude greater than the time to communicate with a remote process, some applications can improve their performance by using non-coherent memory. Furthermore, the combination of a high bandwidth network and a network coprocessor will make the choice of buffer size for the implementation of non-coherent operations a moot issue. This is because writes will not need to be buffered by the process doing the writes. The buffering will be done automatically by the coprocessor which will try to send the messages as fast as it can. It can change the buffer size adaptively depending on the state of the network. The high bandwidth of the network will be able to accommodate the large number of messages that may be generated. The latency of the medium will still be a factor.

Multiprocessor desktop workstations are widely available now. The processors in these systems share memory that is connected to a bus and the hardware protocols guarantee coherence. A parallel computation on a network of such workstations can make use of the fact that within processes on the same workstation the memory is coherent. To describe such computations our model needs to be extended to allow for different semantics of an operation among different groups of processes.

# Bibliography

[ABC+89] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proc. 3rd ACM Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts*, April 1989.

[ABHN91] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proc. of the Fifth International Workshop on Distributed Algorithms*, pages 9–30, Oct. 1991.

[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, College of Computing, Georgia Institute of Technology, 1990.

[AHJ91] Mustaque Ahamad, Philip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proc. 11th IEEE Intl. Conference on Distributed Computing Systems*, June 1991.

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

[BBD+87] J. Boyle, R. Butler, T. Disz, B. Glickgield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, 1987.

[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.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[Bir91] Kenneth P. Birman. The process group aproach to reliable distributed computing. Technical Report TR-91-1216, Computer Science Department, Cornell University, July 1991. Revised January 1993. To appear in Communications of the ACM.

[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.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tannenbaum. Orca - a language for parallel programming on distributed systems. *IEEE Trans. on Software Engineering*, 18(3):190–205, March 1992.

[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.

[BT90] Dimitri P. Bertsekas and John N. Tsitsiklis. A survey of some aspects of parallel and distributed iterative algorithms. Technical Report CICS-P-189, Center for Intelligent Control Systems, Cambridge, January 1990.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, September 1991.

[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.

[CLL90] Jen-Yao Chung, Jane Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transaction on Computers*, 19(9):1156–1173, September 1990.

[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.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Annual International Symposium on Computer Architecture*, June 1986.

[Dun92] Trung Dung. Coherence implies sequential consistency. Personal Communication, October 1992.

[FP89] Brett D. Fleisch and Gerald J. Popeck. Mirage: A coherent distributed shared memory design. In *Proc. 12th ACM Symp. on Operating System Principles, Litchfield Park, Arizona*, ACM Press, December 1989.

[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.

[Goo91] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin-Madison, February 1991. Originally appeared in 1989.

[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.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. 19th Annual International Symposium on Computer Architecture*, Goldwater, Australia, May 19-21 1992.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[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.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Dept. of Computer Science, Yale University, New Haven, CT., October 1986.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. 17th Annual International Symposium on Computer Architecture*, May28-31 1990.

[LLG+92] Daniel Lenoski, James Laudon, 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.

[Mat90] Mathematics and Computer Science Division, Argonne National Laboratory. *Using the BBN TC2000*, June 1990. Available as ANL/MCS-TM-135.

[MF89] Ronald G. Minnich and David J. Farber. The Mether System: Distributed Shared Memory for SunOS 4.0. In *Proceeding of Usenix-Summer 89*, 1989.

[MF90] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *Proc. 10th IEEE Intl. Conference on Distributed Computing Systems, Paris, France*, June 1990.

[Min91] Ronald G. Minnich. *Mether: A Memory System for Network Multiprocessors.* PhD thesis, University of Pennsylvania, 1991.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[San90] Jonathan Sandberg. Design of the PRAM network. Technical Report CS-TR-254-90, Computer Science Department, Princeton University, April 1990.

[Ser90] Dimitrios Serpanos. *Scalable Shared Memory Interconnections.* PhD thesis, Princeton University, October 1990.

[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.