



Distributed Parallel Computing in Mermera: Mixing Noncoherent Shared Memories*

Abdelsalam Heddaya[†]
heddaya@cs.bu.edu

Himanshu Sinha
hsinha@gte.com

Computer Science Dept.
Boston University

GTE Laboratories, Inc.
Waltham, MA

BU-CS-96-005[‡]

March 7, 1996

(Revised April 19, 1996)

(Revised October 3, 1996)

Abstract

Programmers of parallel processes that communicate through shared globally distributed data structures (DDS) face a difficult choice. Either they must explicitly program DDS management, by partitioning or replicating it over multiple *distributed memory* modules, or be content with a high latency *coherent* (sequentially consistent) memory abstraction that hides the DDS' distribution. We present Mermera, a new formalism and system that enable a smooth spectrum of noncoherent shared memory behaviors to coexist between the above two extremes. Our approach allows us to define known noncoherent memories in a new simple way, to identify new memory behaviors, and to characterize generic mixed-behavior computations. The latter are useful for programming using multiple behaviors that complement each others' advantages.

On the practical side, we show that the large class of programs that use *asynchronous iterative methods* (AIM) can run correctly on *slow memory*, one of the weakest, and hence most efficient and fault-tolerant, noncoherence conditions. An example AIM program to solve linear equations, is developed to illustrate: (1) the need for concurrently mixing memory behaviors, and, (2) the performance gains attainable via noncoherence. Other program classes tolerate weak memory consistency by synchronizing in such a way as to yield executions indistinguishable from coherent ones. AIM computations on noncoherent memory yield noncoherent, yet correct, computations. We report performance data that exemplifies the potential benefits of noncoherence, in terms of raw memory performance, as well as application speed.

Keywords: Distributed parallel computing, noncoherent shared memory, asynchronous iterative algorithms, network of workstations.

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

[†]On sabbatical leave at Harvard University's Aiken Computation Laboratory, and Department of Biological Chemistry and Molecular Pharmacology.

[‡]This document's URL is (<http://www.cs.bu.edu/techreports/96-005-mermera-model-system.ps.Z>).

1 Introduction

Writers of parallel programs on a network of workstations face two unpalatable choices when it comes to sharing data among processes. Either they must manage the distribution of data explicitly, or they have to rely on traditional, sequentially consistent, shared memory systems. The latter option has the disadvantage of high latency for some operations, especially on a network of workstations (NOWs) where the access time for remote memory is several orders of magnitude greater than the access time for local memory.

In [26], Lipton and Sandberg showed that the worst case access time for sequentially consistent memory is proportional to the worst case communication delay among the participating processes. In response, several proposals have been made to improve the performance of shared memory systems by relaxing the commonly used correctness condition of sequential consistency. Most systems (see for examples [1, 15, 16, 17]) require that all processes agree on a global sequencing of all write operations to the same location, even though they may vary in their perception of the interleaving order of write operations to different locations. These systems typically require the programmer to identify—via program annotations—the units of execution in terms of whose boundaries memory consistency is defined, and to describe the pattern of memory references made by the program.

Other proposals like *causal* memory [3], *Pipelined Random Access Memory* (PRAM) [26], *slow* memory [23] and *hybrid consistency* [5] admit executions in which there may not be a single, globally recognized, total order on writes to a single location. In general, as the degree of noncoherence increases, more opportunities for minimizing communication and synchronization overheads become available to the memory system designer. One notable example is *slow* memory, which can tolerate unreliable communication, as well as much message reordering by the network. Only asynchronous, unreliable, and unordered message passing systems provide comparable levels of flexibility in optimizing system performance. This paper concentrates on noncoherent memory systems.

Early specifications of noncoherent memories were given in terms of descriptions of algorithms that implement them. This made it hard to compare different behaviors and to reason about the correctness of programs that run on them. In [22] we present a formalism based on partial orders on the events of an execution to describe these memories. Other formalisms can also be found in the literature [3, 6, 27]. These formalisms have been used to prove the correctness of various program classes on different noncoherent memories. Most approaches establish the correctness of executions of certain programs, by showing that each such program induces only sequentially consistent computations on the shared memory system in question. In [29] we prove that asynchronous iterations [9] converge using *slow* memory. This is the first instance of a proof of correctness of an algorithm on noncoherent memories that results in executions that are *not* sequentially consistent.

Different programs, even different parts of the same program, can tolerate different levels of noncoherence. Therefore, it behooves a system to provide different behaviors to the programmer who can then mix and match the behaviors according to the needs of the different algorithms used in a program. In [21, 29] we present a formal description of mixed behaviors which permits executions where operations can correspond to *slow*, *PRAM*, *causal* and *coherent* memories. Attiya *et al.* [6] give a formalism in which one noncoherent behavior is combined with one noncoherent behavior. Other mixed-coherence behaviors are described in [2, 30].

In this paper, we present *Mermera*¹, a formal model and system for specifying, reasoning about, and programming shared memory systems. Our contributions can be summarized as follows.

¹The Latin root for memory, *mer* derives from the the Sanskrit root *smar*. Our coinage, ‘Mermera’, forms the feminine parallel of the ancient Greek name, *Mermeros*, which means care laden.

- Our formal model, based on the one proposed in [20] captures several known noncoherent behaviors and is significantly simpler than the original proposal. It also enables us to define new behaviors.
- The formalism is extended to describe the behavior of systems in which different kinds of memory behaviors can be mixed in a rational manner. By giving these mixed behavior systems an unambiguous description we make it possible to reason formally about them.
- The formalism is used to prove that asynchronous iterations converge when they execute on *slow* memory. Other authors [3, 6] have shown that certain programs (*e.g.* data race free programs) result in sequentially consistent executions even with some types of noncoherent memory. This paper presents a class of algorithms that are correct on *slow* memory even if the resulting execution is not sequentially consistent.
- An example program that uses mixed-coherence to solve a system of linear equations is presented. While *slow* memory is sufficient for the *convergence* of asynchronous iterative methods, *detection* of the termination requires stronger behavior. The Mermera system provides a programming interface that gives programmers access to different types of memory behavior in the same program.
- We describe a memory system for a network of workstations that implements our approach, characterize its raw performance and the performance of the linear equation solver that uses asynchronous iterations. To the best of our knowledge it is the first such implementation. Our example program achieves superlinear speedup when executed on Mermera.

Section 2 of this paper presents a formal model that describes several types of noncoherent memory, and a framework for mixing some of them in a system. In Section 3 the model is used to prove the correctness of asynchronous iterative methods on *slow* memory. Other applications that can tolerate noncoherence are also described. Our prototype that implements a mixed-coherence memory, is discussed in Section 4. Section 5 gives an example program that uses the different behaviors provided by this system to solve a linear system of equations. Section 6 reports performance measurements of this prototype and that of an application, illustrating the potential for dramatic performance improvements achievable by some applications when they run on mixed-coherence memory. Section 7 summarizes related work in this area and we present our conclusions in Section 8.

2 Mermera memory model

We base our model on memory *events* rather than on memory states, since the latter are only observable through the former. An *event* is a particular execution of an operation o , accessing a given shared object $x \in \sigma$, on behalf of a specific process $i \in \pi$. We denote such an event by $i.x.o_b^a$, where a and b represent the lists of arguments and results, respectively. For simplicity, we assume events to be unique. Whenever i is omitted it is understood that it is irrelevant, while dropping x implies that it is fixed in the given context. Thus, when events are denoted simply by w^a or r_b , this means that they apply to the same object, but may belong to the same or to different processes, depending on context. 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. Definitions of each memory behavior, be it *coherent* or otherwise, expresses the constraints on permissible event orderings. The notation and the definitions we need are given in

Table 1: Glossary of notation.

Notation	Meaning
$i.x.o_b^a$	Shared memory event. Object $x \in \sigma$ performs operation o on behalf of process $i \in \pi$, with argument(s) a , and result(s) b .
$E(\varphi)$	The set of all events that satisfy filter φ , e.g., $E(i.*.w \vee j.*.*)$ is the set of all writes by process i and all events by j . When φ is evident from context, we simply write E .
\xrightarrow{i}	Process program ordering imposed by i 's program on its events, $E(i.*.*)$. Must be a partial order.
\xrightarrow{wr}	<i>Writes-to</i> ordering between write events and the read events that read their values (Definition 1).
$H(\varphi) = (E(\varphi), \rightsquigarrow)$	Shared memory history, with ' \rightsquigarrow ' constructed by taking the transitive closure of the union of all process program orderings, and the writes-to ordering (Definition 2).
$\text{level}(e) = l \in L$	Designates the memory coherence level l of event e , for purposes of defining correctness of histories with mixed coherence. L must be totally ordered so that $l_1 \preceq l_2$ iff coherence level l_1 is stronger than, or equal to, that of l_2 .

Table 1. For simplicity, we assume that no two write events *to the same location* store the same value, i.e., $w^a \neq w^b \Rightarrow a \neq b$. This assumption can be removed by attaching, to each value stored, a unique sequence number that can be easily assigned by the device that issues the write operation.

A *process* i is a set of events, together with an irreflexive partial ordering relation ' \xrightarrow{i} ' such that $e \xrightarrow{i} e'$ iff process i 's program requires that e precede e' . The key ordering induced by the shared memory itself, that contributes to the definition of a history, consists of the writes-to relation ' \xrightarrow{wr} ', which relates a write w^a to every read r_a that reads the value it stores.

Definition 1 A write event w^a *writes-to* every read event r_a that returns the value it stored, i.e., $\forall w^a, r_a : w^a \xrightarrow{wr} r_a$. ■

A shared memory history is an irreflexive transitive ordering ' \rightsquigarrow ' on a set of events, that captures the notion of an event observing the effects of another either directly or indirectly. This enables us to define properties of shared memory histories via constraints on what processes can observe.

Definition 2 A *shared memory history* $H(\varphi)$, of events satisfying φ is an ordering $(E(\varphi), \rightsquigarrow)$, where, $\forall e, e' \in E(\varphi) : e \rightsquigarrow e'$ iff $e \xrightarrow{i} e' \vee e \xrightarrow{wr} e' \vee e \rightsquigarrow^n e'$, where $n \geq 2$. ■

By " $e \rightsquigarrow^n e'$, $n \geq 2$," we mean that there exists a sequence of distinct events e_1, e_2, \dots, e_{n-1} such that,

$$e \rightsquigarrow e_1 \rightsquigarrow e_2 \rightsquigarrow \dots \rightsquigarrow e_{n-1} \rightsquigarrow e',$$

and our intent is to cause the ordering being defined to be transitive.

In order to reason about shared memory histories, we identify two relations, called *overwritten-by* and *reads-overwritten-value* designed to capture orderings that threaten to break the memory behaviors of interest. Each of these relations introduces a sort of *cross* or *back edge* that, when combined with other orderings, can form a cycle that breaks some partial ordering requirement. Both relations deal with a situation when a read event observes a write event to have “overwritten” another. We adopt a style of definition of memory system behavior that can be used directly to decide whether a given shared memory history satisfies a particular property.

2.1 Memory coherence

The following “overwrite” relation helps capture histories that can break coherence (or sequential consistency).

Definition 3 A write event w^a is overwritten by w^b , i.e., $w^a \xrightarrow{ww} w^b$, iff $\exists r_b : w^a \rightsquigarrow r_b$. ■

Definition 4 A shared memory history (E, \rightsquigarrow) is coherent if and only if (E, \overrightarrow{CO}) is a partial ordering, where $e \overrightarrow{CO} e'$ iff $e \rightsquigarrow e' \vee e \xrightarrow{ww} e' \vee e \overrightarrow{CO}^n e'$, where $n \geq 2$. ■

This new style of defining memory coherence, first proposed in [20] by us, appears at formal variance with Lamport’s original and widely used *sequential consistency* condition [25], which can be paraphrased as follows.

Definition 5 A shared memory history (E, \rightsquigarrow) is sequentially consistent (SC) if and only if there exists a total ordering (E, \overrightarrow{SC}) , such that:

1. $e \xrightarrow{i} e' \Rightarrow e \overrightarrow{SC} e'$. The total ordering must preserve program ordering.
2. $\forall w^a, r_a : w^a \overrightarrow{SC} r_a \wedge \nexists w^b : w^a \overrightarrow{SC} w^b \overrightarrow{SC} r_a$. Every read returns the value written by the most recent write in the total ordering. ■

However, both definitions are equivalent, as demonstrated by the proof of the following theorem. The distinguishing feature between *coherence* and *sequential consistency* has to do with proving that a history satisfies the condition. To prove a history sequentially consistent, one needs to establish the existence of a total order ‘ \overrightarrow{SC} ’ with the requisite properties. In contrast, proving *coherence* by our definition necessitates only proving that the constructively defined ‘ \overrightarrow{CO} ’ is a partial order.

Theorem 1 A shared memory history (E, \rightsquigarrow) is sequentially consistent if and only if it is coherent.

Proof. (SC implies CO; only if part.) We first assert and prove two propositions that relate ‘ \overrightarrow{SC} ’, which must exist by the premise of this part of the proof, and ‘ \overrightarrow{CO} ’.

$$(I) \quad e \rightsquigarrow e' \Rightarrow e \overrightarrow{SC} e'.$$

Proof. By rules 1 and 2 of Definition 5 of SC, $e \xrightarrow{i} e' \Rightarrow e \overrightarrow{SC} e'$, and, $e \xrightarrow{wr} e' \Rightarrow e \overrightarrow{SC} e'$, respectively. Therefore, $e \rightsquigarrow e' \Rightarrow e \overrightarrow{SC} e' \vee e \rightsquigarrow^n e'$, $n \geq 2$. *Q.E.D.*

$$(II) \quad w^a \xrightarrow{ww} w^b \Rightarrow w^a \overrightarrow{SC} w^b.$$

Proof. From Definition 3 of the overwritten-by relation, we have, $w^a \xrightarrow{ww} w^b \Rightarrow \exists r_b : w^a \rightsquigarrow r_b$. Therefore, $w^a \overrightarrow{SC} r_b$, by Proposition I above. The fact that ‘ \overrightarrow{SC} ’ is a total order forces either $w^a \overrightarrow{SC} w^b$, in which case are done, or $w^b \overrightarrow{SC} w^a \overrightarrow{SC} r_b$, which contradicts Definition 5, rule 2. *Q.E.D.*

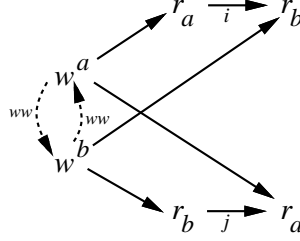


Figure 1: A history that is *causal* but not *coherent*.

Assume by way of contradiction that a sequentially consistent history is not *coherent*. Given that ' $\xrightarrow{C_o}$ ' is transitive by construction, then it must fail to be anti-symmetric, *i.e.*, $\exists e, e' : e \xrightarrow{C_o} e' \wedge e' \xrightarrow{C_o} e$. By Definition 4, and Propositions I and II above,

$$\begin{aligned} e \xrightarrow{C_o} e' &\Rightarrow e \rightsquigarrow e' \vee e \xrightarrow{ww} e' \Rightarrow e \xrightarrow{S_c} e', \text{ and} \\ e' \xrightarrow{C_o} e &\Rightarrow e' \rightsquigarrow e \vee e' \xrightarrow{ww} e \Rightarrow e' \xrightarrow{S_c} e. \end{aligned}$$

Thus, ' $\xrightarrow{S_c}$ ' is not anti-symmetric, which contradicts the premise that it is a total order.

(CO implies SC; if part.) First, we note that ' $\xrightarrow{C_o}$ ' being a partial order, implies that its subset, ' \rightsquigarrow ', is also a partial order. Let ' \xrightarrow{T} ' be an irreflexive total order resulting from augmenting the partial ordering ' \rightsquigarrow ' arbitrarily; we will show that it satisfies the definition for ' $\xrightarrow{S_c}$ ', or can be so modified. From the definition of *coherent* shared memory, we have $e \xrightarrow{i} e' \Rightarrow e \rightsquigarrow e' \Rightarrow e \xrightarrow{T} e'$, which satisfies the first condition of sequential consistency.

We prove the second condition by contradiction. Assume that $\exists w^a \xrightarrow{T} w^b \xrightarrow{T} r_a$. If $w^b \not\rightsquigarrow r_a$, then we can modify ' \xrightarrow{T} ' so that $r_a \xrightarrow{T} w^b$. Similarly, if $w^a \not\rightsquigarrow w^b$, then we can modify ' \xrightarrow{T} ' so that $w^b \xrightarrow{T} w^a$. Otherwise, it must be that $w^a \rightsquigarrow w^b \rightsquigarrow r_a$. But, $w^a \rightsquigarrow w^b \Rightarrow w^a \xrightarrow{C_o} w^b$, and $w^b \rightsquigarrow r_a \Rightarrow w^b \xrightarrow{ww} w^a \Rightarrow w^b \xrightarrow{C_o} w^a$, which contradicts the premise that ' $\xrightarrow{C_o}$ ' is a partial order. ■

2.2 Memory noncoherence

Processes can be said to agree on the ordering of events if their individual observations of these events induce *compatible* orderings. If we consider orderings compatible when they do not contradict each other at all, we have *coherent* memory; but if we permit them to disagree over the ordering of concurrent events we obtain *causal* memory. In other words, *coherent* memory requires processes to have compatible perceptions of the ordering of even unrelated *writes*, rejecting the history shown in Figure 1, for example. By contrast, *causal* memory [3], defined below, enables processes to disagree on the ordering of such concurrent write events. Interestingly, the notion of *potential causality* first formalized by Lamport [24], to describe the partial ordering induced by point-to-point messages is akin to *coherent* memory, not *causal* memory. Shared memory systems in general correspond to multicasting message passing systems, such as Isis [12], since values written to a single memory location may be read by multiple processes, that can then disagree over the order of values written by causally unrelated writes. This is not the case for point-to-point messages, where causally unrelated message *sends* cannot be ordered differently by different processes, because only one process can perceive a *send* event. In the rest of this section, we define a variety of noncoherent memories, starting with *causal*.

The role played by the overwritten-by relation in defining coherence, is played by a different relation in the case of noncoherence.

Definition 6 A read r_a reads an overwritten value witnessed by event e , in which case we say $r_a \xrightarrow[r/e]{} e$, iff $\exists e' : e \rightsquigarrow e' \rightsquigarrow r_a$, where $e \in \{w^a, r_a\}$, and $e' \in \{w^b, r_b\}$. ■

For example, this relation can be induced when a *write* w^a is overwritten, yet is still subsequently observed by r_a , i.e., $w^a \rightsquigarrow r_b \rightsquigarrow r_a$, and hence $r_a \xrightarrow[r/e]{} w^a$. In another scenario, $r_a \xrightarrow[r/e]{} r_a$ arises from $r_a \rightsquigarrow w^b \rightsquigarrow r_a$, which means that some write w^b was overwritten by a *preceding* write. This relation is amply illustrated in Figures 2–6.

Definition 7 A shared memory history $H = (E, \rightsquigarrow)$ is causal iff $(E, \xrightarrow[\overline{C}]{})$ is a partial ordering, where $e \xrightarrow[\overline{C}]{} e'$ iff $e \rightsquigarrow e' \vee e \xrightarrow[r/e]{} e'$. ■

This definition for *causal* memory histories, agrees with the original informal proposal by Hutto and Ahamad [23] in 1990, and with its later formalization by us [20] in 1992, and subsequently by others [3, 2].² In the context of shared memory, coherence implies causality, but not *vice-versa*. The following theorem establishes the former statement, while Figure 1 provides an example that ensures that the set of *causal* histories strictly includes that of *coherent* histories.

Theorem 2 If a shared memory history $H = (E, \rightsquigarrow)$ is coherent then it must also be causal.

Proof. It is straightforward to prove the contrapositive, i.e., that if H is not *causal* then it cannot be *coherent*. We leave the proof to the reader, noting only that it hinges on showing that $r_a \xrightarrow[r/e]{} e$ implies that $\exists w^a, w^b : w^b \xrightarrow[ww]{} w^a \wedge (w^a \rightsquigarrow w^b \vee w^a \xrightarrow[ww]{} w^b)$. ■

While *coherence* and *causality* require that processes agree on the ordering of causally related events in the entire global history, additional noncoherence arises when processes are allowed to disagree on portions of the global history. For example, Lipton and Sandberg’s [26] *pipelined random access memory* (PRAM) differs from *causal* memory in requiring that processes agree only on the ordering of write events issued by the same process. Similarly, Hutto and Ahamad’s [23] *slow* memory stipulates that processes agree on the order of write events issued by the same process to the same location. Writes to different locations by a process may be observed in different orders by different processes. Table 2 defines these two memories, as well as three new ones, based entirely on the causality of subhistories. This represents a new rationalization of the memory space that opens new possibilities, which we have only begun to explore. Figures 2–4 give examples to illustrate and contrast some of the defined behaviors.

Our definitions of memory behaviors immediately yield the inclusion relationships shown as a lattice in Figure 5, and the examples in Figures 1–4 establish the strictness of the inclusion, e.g., *coherent* \subset *causal* \subset *PRAM* \subset *slow* \subset *local*. Total orders in the memory behavior lattice play an especially important role in enabling a clean and correct definition of mixed memories, as shown in Section 2.3 below. For space and scope considerations, we do not discuss the other two local behaviors, *object local* and *process local*.

Aside from our new definition of *coherence*, the novelty in our formalism arises in our definitions of *PRAM* and *slow* memories. Our approach to the construction of these definitions has three new features, compared to other work in the field:

²Definition 7 is actually simpler than our 1992 one.

Table 2: Noncoherent memory definitions.

A shared memory history H is	if and only if
<i>PRAM</i>	$H(i.*.w \vee j.*.*)$ is <i>causal</i> , $\forall i, j \in \pi$.
<i>Slow</i>	$H(i.x.w \vee j.x.*)$ is <i>causal</i> , $\forall i, j \in \pi, \forall x \in \sigma$.
<i>Process local</i>	$H(i.*.*)$ is <i>causal</i> , $\forall i \in \pi$.
<i>Object local</i>	$H(*.x.*)$ is <i>causal</i> , $\forall x \in \sigma$.
<i>Local</i>	$H(i.x.*)$ is <i>causal</i> , $\forall i \in \pi, \forall x \in \sigma$.

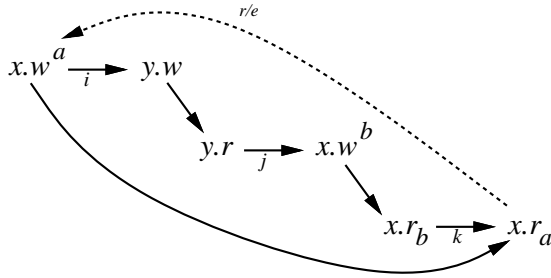


Figure 2: A history that is *PRAM* but not *causal*.

1. It is comprehensive. This enables us to identify three new memory conditions, *object local*, *process local* and *local* that have not been discussed before. The first of these is likely to be of special interest since it is stronger than *slow* memory, and hence can run all programs that tolerate that level of noncoherence (see Section 3 below). In addition, *object local* coherence enables object-oriented implementations.

Our formalism clearly suggests further unexplored variations, such as basing the lattice on *coherence* instead of *causality*, which we are currently investigating.

2. It is simple. We introduce two intuitive relations: *overwritten-by* and *reads-overwritten-value* witnessed by event e , that suffice to capture a wide range of potential anomalies threatening to break different levels of coherence. Different memory behaviors then arise from straightforward composition of a small number of elementary constructions.
3. Most importantly, the above two advantages come together to help us formalize the correctness condition of *generic* mixed memory histories (see Section 2.3).

2.3 Mixed levels of coherence

We extend the formalism described in Sections 2.1 and 2.2 above to characterize histories that permit the *concurrent mixture* of behaviors in Mermera. This extension takes into account the fact that a subhistory obeying a particular property, say *PRAM*, can be affected by the information that may flow through weaker subhistories involving *slow* events. In other words, applying the preceding definitions to naively constructed subhistories would not suffice. To show that this is the case, consider the example history shown in Figure 6. If we simply label individual events with the

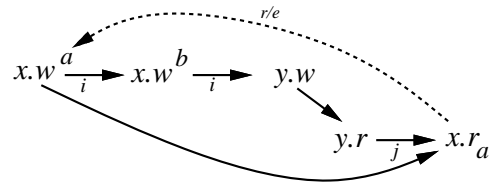


Figure 3: A history that is *slow* but not *PRAM*.

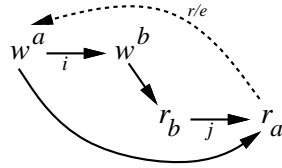


Figure 4: A history that is *local* but not *slow*.

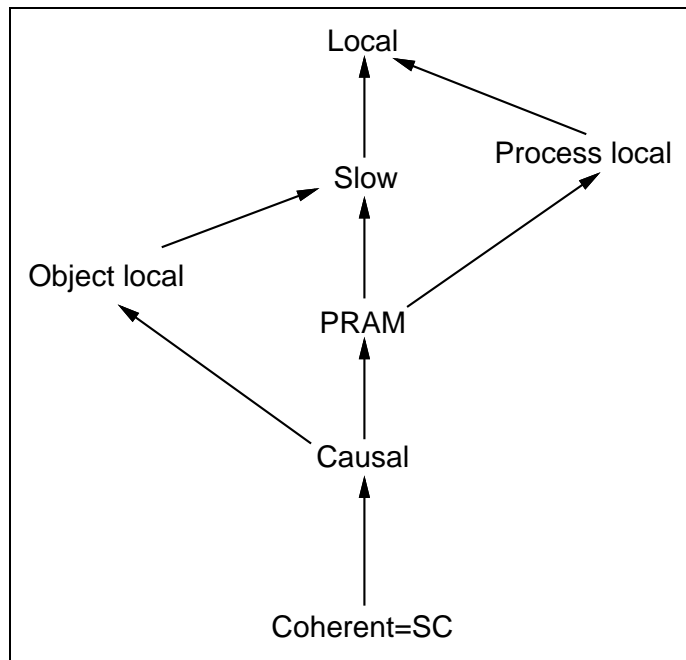


Figure 5: Lattice of memories defined in this paper.

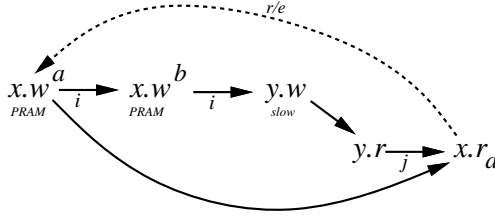


Figure 6: A history that is not *mixed-PRAM* according to our definition, but that would qualify as *PRAM* in a naive definition that excludes *slow* events.

level l of coherence they must satisfy, and apply our definitions to subhistories consisting of events belonging to l or lower, then we may admit histories that should be rejected. The figure shows a history that should not qualify as *mixed-PRAM* because of information that has flowed through a *slow* write event.

Every event e has a label, $\text{level}(e)$, that identifies its coherence level, drawn from a *totally ordered* set L of memory behaviors. The reflexive total order \preceq on L models the relationship of a set of histories being included in another, or, equivalently, the relationship of one memory behavior being strictly stronger than another. This means that valid choices of L must consist of memory behaviors that form a strictly linear hierarchy, with each behavior being admitted by all the weaker ones.

For example, our Mermera *system* described in Section 4 below implements $L = \{\text{coherent}, \text{PRAM}, \text{slow}, \text{local}\}$, and allows write events to be labeled freely with any level in L . Read events, by contrast, are all considered *coherent*, and hence not explicitly labeled in our system.³

We have two choices for defining correctness of histories with mixed levels of coherence: either constrain the sets of events that must satisfy each level in L to events labeled by that level or stronger, or, limit the definition of event ordering relations so as to include only event pairs labeled appropriately. The latter approach has the advantage of enabling us to respect information flowing through weaker events, when constructing relations among events at a particular coherence level. Our generalization of the definitions of overwrite relations (\xrightarrow{ww} and $\xrightarrow{r/e}$), excludes from consideration, for coherence level l , any ordered event pair that has at least one member that is not at level l or stronger. However, it admits pairs that are ordered as a result of information flowing through events that may be weaker.

Definition 8 A *labeled overwrite relation*, $e \xrightarrow{\rho} e'$, holds iff $e \xrightarrow{\rho} e' \wedge \text{level}(e) \preceq l \wedge \text{level}(e') \preceq l$, where $\rho \in \{ww, r/e\}$. ■

We say that a labeled history H is *mixed-causal*, for instance, iff it satisfies Definition 7, after substituting $\xrightarrow{C_a}$ for \xrightarrow{ww} . Similarly, we generalize the definitions in Table 2. A mixed history H , whose events are all labeled as above, is then considered *correct* if and only if, for every label $l \in L$, history H is *mixed- l* .

The notion of mixed coherence levels turns out to be necessary for the correct exploitation of noncoherence for performance, as illustrated by the programming example in Section 5.

³In the model of mixed consistency proposed in [2], Agrawal *et al.* choose a dual scheme, where reads are labeled freely, but writes are not. We believe this to be a system design choice that should not be frozen in the model.

3 Programs that tolerate noncoherence

In general, a program—or a fragment thereof—can tolerate noncoherence in three situations. One, if the program’s intrinsic synchronization, coupled with the appropriate noncoherent memory condition, yields executions that are coherent. Examples of such program classes include *data-race free* programs [3, 6], which run as coherent on *causal* memory. Two, a program that can detect inconsistency, can run well on a very weak memory, until such inconsistency arises, then switch to coherent memory behavior to eliminate it. For instance, certain parallel B-tree algorithms are able to detect inconsistencies in the tree structure that result from concurrent modification [30], and hence are amenable to execution on an especially weak multiversion memory. The third type of programs, of which no instance was previously known, covers programs that can function correctly with a noncoherent execution that is not equivalent to a coherent one. The following subsection identifies a large class of this third kind, and proves that it runs correctly on one of the weakest known noncoherent memories. Subsection 3.2 contains a detailed discussion of programs of all three types.

3.1 Asynchronous Iterative Methods

In this section we show that under certain conditions *slow* memory is sufficient for the correctness of *totally asynchronous iterative algorithms* (AIM), the importance of which for parallel computing was first recognized by Baudet [7]. Bertsekas and Tsitsiklis give a comprehensive discussion of AIM in [9, 10]. Such algorithms find a fixed point $a = f(a)$ of the iteration $x \leftarrow f(x)$, 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, denoted by P_i , computes x_i .

Let $x_j(\tau_j^i(t))$ be the value of x_j available to P_i at time t . P_i computes $x_i(t+1)$ as follows:

$$x_i(t+1) \leftarrow f_i(x_1(\tau_1^i(t)), x_2(\tau_2^i(t)), \dots, x_n(\tau_n^i(t))), \quad 0 \leq \tau_j^i(t) \leq t,$$

where $\tau_j^i(t)$ denotes the “version” of x_j used by P_i in computing $x_i(t+1)$.

Bertsekas and Tsitsiklis show in [10] that if the instance of the problem satisfies certain conditions and the *total asynchrony* assumption is satisfied then the iteration described above will converge. The total asynchrony assumption states that:

1. The iteration $x \leftarrow f(x)$ is repeated infinitely often, and,
2. If $\{t_k\}$ is an increasing time sequence that tends to infinity then $\lim_{k \rightarrow \infty} \tau_j^i(t_k) = \infty$.

We will now show that under certain conditions the total asynchrony assumption is satisfied by a system using *slow* memory to store x .

Theorem 3 *If an asynchronous iterative program converges under the total asynchrony assumption, then it also converges under slow memory with some liveness guarantee.*

Proof. We need only show that *slow* memory satisfies the above total asynchrony assumption. The first condition, that the iteration be repeated infinitely often, can be satisfied by the program implementing this iteration. A simple infinite loop that keeps computing f_i satisfies this condition.

To prove the second condition we first show that $\tau_j^i(t+1) \geq \tau_j^i(t)$. We prove this by contradiction. Assume that there exist i, j and t such that $\tau_j^i(t) > \tau_j^i(t+1)$. Let $\tau_j^i(t) = t_1$ and $\tau_j^i(t+1) = t_2$. An execution that allows this contains the history segment shown in Figure 7. This is not permitted by *slow* memory because the order imposed by ‘ $\frac{t}{r/e}$ ’ violates the definition. This shows that $\tau_j^i(t)$

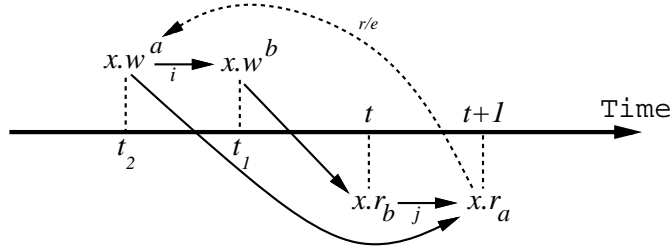


Figure 7: A computation in which $t_1 = \tau_j^i(t) > \tau_j^i(t+1) = t_2$.

is a monotonically non-decreasing function. If an implementation of *slow* memory also guarantees progress (or liveness), in the sense that, either a write by j eventually reaches i , or a subsequent write by j does, then $\tau_j^i(t)$ is guaranteed to increase eventually. Since x_j is written infinitely often (because of Condition 1), $\lim_{t \rightarrow \infty} \tau_j^i(t) \rightarrow \infty$. ■

Examples of iterations that converge when the total asynchrony assumption is satisfied include:

1. A linear system of equations, $x \leftarrow Ax + b$, such that the spectral radius of A , $\rho(|A|) < 1$. [7]
2. Some graph algorithms, *e.g.*, Bellman-Ford all pairs shortest path algorithm [8].
3. All dynamic programming algorithms.

A comprehensive list of fixed point problems that converge in a totally asynchronous iteration can be found in [9].

3.2 Other Program Classes

In Section 2.2 we established the lattice shown in Figure 5. 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 lattice. But, *which of these memories are useful to programmers?* Memories higher in the lattice 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 lattice.

Weak memory—whose histories can be defined to obey the truly “weak” condition: $\forall r_a \in E : \exists w^a \in E$ —would be the most efficient to implement. Multi-version memory (MVM), proposed by Wang and Weihl [30] to support highly concurrent B-trees, is a memory in which the programmer has access to a variant of *weak* memory, as well as to *dynamic atomic* memory. The former is constrained beyond the above definition by the particular implementation, and the latter is *coherent* memory, with an additional ordering induced by the real time sequence of event intervals (duration between invocation and return). The program uses *weak* memory, switching to *dynamic atomic* only when it detects an inconsistency. Such detection is achieved through redundancy introduced in the B-tree data structure itself.

In [23], 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. They also argued that one cannot achieve mutual exclusion using *slow* Memory. Attiya and Friedman [5] 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.

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 3.1 that *slow* memory with some liveness guarantee 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.

Lipton and Sandberg show in [26] 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*⁴. They also prove that in these computations, whenever the cost of the computation dominates the synchronization overhead⁵, *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 [4]. A complete proof that every *data-race free* program executes on *causal* in a manner that yields a sequentially consistent behavior later appeared in [3].

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.

4 Mixed coherence in the Mermera System

In Section 3 we discussed a variety of application classes that tolerate different degrees of noncoherence. This section describes the Mermera system which permits programmers to use different types of memory operations in a program.

4.1 Architectural Choices

Fixed vs dynamic determination of degree of (non)coherence: A system can require a location to be of a particular type of memory and this type would be fixed throughout the program. Thus, a location, x , may be specified to be *slow* and another location, y , may be specified to be *coherent*. The memory types of x and y cannot be changed in the program.

Alternatively, a system can allow programmers to use different types of operations on the same location. This would permit programmers to use whatever behavior they feel is appropriate at any point in the program. Section 5 presents an algorithm that uses different levels of coherence for the same location in the same program.

The choice of a fixed behavior for a location would make a program more readable and therefore easier to reason about. But different parts of a program may tolerate different levels of noncoherence. If the behavior is fixed for the whole program then the programmer would have to choose the strongest (i.e., the most coherent) behavior as the type of a location.⁶ On the other hand, the choice of dynamic behavior for a location allows programmers to extract the best performance in each part of a program. Mermera allows different behaviors for a location in a program.

⁴“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.” [26]

⁵Logical synchronization by the program, *not* by the memory.

⁶Programmers could get around this drawback by creating new locations for each part of the programs and copying values from old locations into the new locations. The new locations would support the behavior appropriate for that part of the program. However, this technique is quite cumbersome.

Which types of memory to support? Our model characterizes the mixing of different types of noncoherent memories with coherent memory. In Section 2 we described a correctness condition for mixed-coherence systems based on a total order on the different types of memories (Figure 5). A system designer is faced with the decision of which of these behaviors to support. Coherent memory must be supported because most shared-memory parallel algorithms have been proven correct on coherent memory. The decision of which noncoherent memories to support would primarily be based on level of noncoherence that the target applications can tolerate and the extent to which they can benefit from the varying degrees of noncoherence. Section 3 discusses applications that tolerate different levels of noncoherence.

In our implementation, *slow* behavior is chosen because it is the weakest behavior that is sufficient for the convergence of asynchronous iterative methods. PRAM is chosen because it is the weakest behavior for which a liveness condition makes sense. The liveness condition satisfies the progress requirement needed for the convergence of asynchronous iterative methods. The *locally consistent* behavior allows programmers to use shared memory as private memory and operations of this kind are inexpensive to implement. These behaviors are also easy to implement with the tools currently available, as seen in Section 4.3 below. *Causal* behavior is omitted because existing algorithms for its implementation [3] require the significant overhead of the transmission of vector timestamps.

4.2 Programming Interface of Mermera

In this section we describe the programming interface presented by the Mermera system. Mermera mixes the behaviors of coherent memory, pipelined RAM, slow memory and locally consistent memory. Processes in the system share a region in their address spaces with other processes that may be on a different processor.

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. The behavior of the Mermera system is formally defined by the correctness conditions for mixed-coherence discussed in 2.3. The description below gives an equivalent intuitive notion of the different noncoherent behaviors.

CO_Write(*loc*, *val*): This operation provides the behavior specified by *coherent* memory, *i.e.*, all *CO_writes* are totally ordered.

PRAM_Write(*loc*, *val*): This operation provides the behavior specified by *PRAM*. 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_Writes* 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*.

The above informal semantics of Mermera's operations is made perfectly precise in Section 2 above. In particular, the mixed behavior is formalized and explained in Section 2.3.

4.3 Implementation

We employ an update-based protocol that optimizes read operations by making a copy of the shared memory at each process. A read immediately returns the value of a variable in the local copy. A write returns as soon as it can be assured that the ordering constraints imposed by the corresponding coherence level will be satisfied. As a result, noncoherent writes tend to be far faster than coherent writes on a network of workstations. Latency for noncoherent writes tends to be small since it is possible for them to return after performing only local work. Throughput is also improved for noncoherent writes because they can be buffered together, and sent in aggregate so as to extract maximum communication bandwidth from the network. Clearly, many different design approaches are possible, and we do not claim superiority for our design; our purpose is simply to establish the *potential* performance advantages of noncoherence, not to prescribe an implementation method. In the remainder of this section, we describe the implementation in detail, and follow it by a discussion of our design rationale.

Our algorithms rely on full replication, and are update-based, *i.e.*, each process has a copy of the shared memory and this copy is updated as writes occur. A read operation returns the value in the local copy. A write operation updates the local copy and propagates the value to other processes running on Sun Sparc 1+ workstations networked via 10 Mb/s Ethernet. The exact manner of transmission depends on the type of the write operation.

The specification of Mermera does not require that all writes be propagated to other processes. Only *CO_Writes* and *PRAM_Writes* are guaranteed to be propagated to all processes. *Slow_Writes* can be transmitted on a best-effort basis, *i.e.*, the system tries to propagate them but no guarantees are given, other than that the resulting execution does not violate the formal definition of *slow* memory. *Local_Writes* are applied only to the local copy, although their specification permits their propagation.

We use version 2.2.5 of the Isis toolkit [12, 11] to propagate the values written to memory, because Isis gives us a suite of group multicast primitives that satisfy relevant ordering properties. The broadcasts of interest to us are *abcast*, *fbcast* and *mbcast*. These primitives obey different constraints on the order in which the messages are delivered to their destinations. *All* messages sent using *abcast* are delivered in the same order at all destinations, *i.e.*, the order in which these messages are delivered is the same for all processes. This is exactly the property we want for *CO_Write*. The *fbcast* messages obey a weaker constraint: messages sent by the *same* process are delivered to all processes in the order they were sent. However, *fbcasts* sent by different processes may be interleaved in different orders at different recipients. This suffices for *PRAM_Writes*, so we employ *fbcasts* to propagate them. No ordering constraints are guaranteed among *mbcast* messages, which we use to transmit *Slow_Writes*.

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

Another feature of Isis that we exploit is its lightweight task system. This allows us to have several concurrent tasks in a user process, each created to handle one received message, which, in turn, names the procedure that the handler should execute. These tasks are scheduled in FIFO order, but non-preemptively, which makes synchronizing accesses to shared data structures very easy, *i.e.*, we do not have to worry about enforcing mutual exclusion on accesses to data structures.

We now explain how we mix updates obeying different levels of coherence. All writes other than *CO_Writes* are buffered until the buffer size exceeds a certain adjustable threshold, or a timeout expires. The (*Location*, *Value*) pair of a *CO_Write* is appended to the buffer and the entire buffer

is immediately broadcast using the *abcast* protocol to all processes (including the writer). The task that issues the operation continues only after the message has been delivered to, and processed by, its own process. This ensures that the *CO_Write* is applied in the global order of *abcasts*. In case of *PRAM_Writes* and *Slow_Writes* the local copy is immediately updated and the updates are appended to the buffer. This buffer is broadcast when it fills up or when a *CO_Write* needs to be broadcast. It is also broadcast when a preset timeout expires. If the buffer is not full then the task issuing the *PRAM_Write* or *Slow_Write* can continue immediately after the local copy has been updated. The buffering of the noncoherent writes allows the cost of their propagation to be amortized over several writes. Further, it permits the overlapping communication with computation because the buffers can be propagated asynchronously.

The multicast primitive used to broadcast a buffer depends on the type of the strongest write in it, according to the hierarchy in figure 5, 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*.

Liveness Requirements. The description of Mermera above 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 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 require that the receiving process become aware of the lost write.

5 Using Mermera

In Section 3, we reviewed the various classes of programs that are known to run safely using noncoherent memory. As a rule of thumb, the programmer need not consider noncoherence except for performance bottlenecks whose speed is limited by memory performance. In trying to optimize the performance of these program fragments, the programmer should first check if his algorithm is known to tolerate noncoherence, in which case, very minor modifications are needed. The modifications in question range from identifying synchronization accesses to the system, as required by some relaxed coherence systems such as *release consistency* [13], and giving hints on the memory reference pattern. The Mermera programmer, by contrast, checks to see if the critical program fragments employ—or can be replaced with—algorithms that tolerate noncoherence. By allowing a well-defined mixed execution using different levels of coherence in the same program, Mermera directly supports the selection of the noncoherence condition that best fits the nature of the performance-critical portions of the user’s program. The remainder of this section is devoted to explaining, through an example program, how to implement an iterative linear equation solver to run on Mermera.

The example program implements an asynchronous iterative algorithm [9] to solve a linear system of equations $Ax + b = 0$, using noncoherent memory on p processes. A is an $n \times n$ matrix, x and b are vectors of size n .

The program shown in Figure 8 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

```

Epsilon = 0.0001                                     /* Accuracy desired */
do
{ do
  { AbsoluteDiff = 0;
    for (i = MyLow; i < MyHigh; i++)
      { NewXi =  $-(b_i + \sum_{j=1}^{i-1} a_{ij} \times x_j + \sum_{j=i+1}^m a_{ij} \times x_j)/a_{ii}$ ;           /* Use Read to read  $x_j$  */
        AbsoluteDiff = AbsoluteDiff + abs(NewXi - Read(XStartLoc + i));
        Slow_Write(XStartLoc + i, NewXi);
      }
    }until (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 */
  }until (global_termination());

```

Figure 8: A linear equation solver.

local termination it *PRAM_Writes* its latest values and then 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 definition of *slow* memory permits a Mermera’s implementation to propagate *Slow_Writes* 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 3.1 as a condition for convergence. In this application, a process does $n - 1$ *read* operations on shared memory to compute each x_i . Therefore, an implementation of Mermera that makes *read* operations fast is ideal for this application.

6 Performance

In this section we summarize the results of two types of experiments with our implementation of Mermera. First, we measure the memory *access time* and *completion time* (defined in the following subsection). Second, we measure the performance of the equation solver described in Section 5 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.

6.1 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. This does not imply that when the write returns, the value written has

been propagated to all processes⁷ sharing memory. A *CO_Write* returns after its position in the global order of all writes is determined. Non-coherent writes return after the local copy is updated and the write is enqueued in a buffer. If the buffer fills up all writes in it are passed on to Isis for asynchronous propagation. To measure the time taken for the operation to be invoked and the value written to be received by all other processes we use a metric called *completion time*. This is the time taken for each process to execute 100 writes in parallel and for the propagation of all these $100p$ values to all 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 hold the noncoherent writes (1, 10, 100, 1000 location-value pairs). For each parameter setting we run the experiments over 100 times and we report the fastest time measured.

The amount of work done for noncoherent writes may be different each time the operation is invoked. This is because if a write causes the buffer to fill up then the propagation of the buffer has to be initiated before the writing process can continue. So, we measure the access time for 100 consecutive writes for each setting of the parameters. Each process issues its writes concurrently with the others. The access times observed by each process may be different from that observed by other processes. This is especially true for *CO_Writes* which rely on totally ordered atomic multicasts. 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 any remote communication to determine their position in the total order. The total access time of 100 writes averaged over all processes 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 is reported.

Measurements

Our measurements are summarized in Tables 3 and 4. The minimum latency of *PRAM_Writes* and *Slow_Writes* are $6.6\mu s$ and $6.2\mu s$ respectively and this is independent of the number of processes involved. This is the time it takes for the write to be buffered for later propagation. The propagation occurs when the buffer fills up or a configurable timeout happens.

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 of 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 coalesces these small messages to larger messages for more efficient transmission. This explains the observation that for a given number of processors the completion time grows at a rate *sub-linear* in the number of messages sent.

Table 4 shows that for a fixed buffer size the completion time of noncoherent memories depends linearly on the number of processes sharing memory because multicasts are carried out by $p - 1$ point to point messages. On the other hand the completion time of coherent writes grows faster

⁷In all our experiments we have one process from each workstation participating in the shared memory computation. Therefore, we use the terms “process” and “processor” interchangeably.

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	0.66	46.9	5.5	1.2	0.62
2	532.0	108.0	16.8	2.5	0.66	96.4	16.1	2.4	0.62
3	1,005.0	101.3	26.6	3.9	0.66	80.6	25.9	3.9	0.62
4	2,049.0	110.8	28.1	5.2	0.66	86.8	27.8	5.1	0.62
5	3,561.0	117.8	27.7	6.3	0.67	86.4	27.2	6.3	0.63
6	5,378.0	120.5	28.0	7.1	0.66	91.4	27.1	7.1	0.62

Table 3: Access time in milliseconds for 100 writes. Buffer Size is measured in terms of the number of write operations that can be buffered.

Procs	CO_Write	PRAM_Write			Slow_Write		
		Buffer Sizes			Buffer Sizes		
		1	10	100	1	10	100
1	136.8	50.4	9.4	4.8	46.8	9.0	4.7
2	1,030.0	267.2	54.2	26.2	245.3	53.9	27.1
3	1,574.0	412.9	100.0	52.4	383.4	99.7	53.9
4	2,889.0	638.0	148.0	85.8	601.4	145.8	85.7
5	4,645.0	963.8	188.4	124.9	871.9	187.9	125.4
6	6,761.0	1204.0	247.9	170.9	1223.0	246.8	174.0

Table 4: Completion time in milliseconds for 100 writes.

than linearly as the number of processes is increased. Completion times for *CO_Writes* is 3-38 times slower than the noncoherent writes. The factor increases as the number of processes increases.

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.

6.2 Solving a System of Equations

In this section we present the performance of the linear equation solver given in Section 5. The performance of a version that uses the noncoherent behaviors of Mermera is compared with another version that uses coherent behavior only. The effect of buffer size on the performance of the solver is discussed.

Experimental Methodology

In Section 5 we presented a program to solve a linear system of equations, $Ax + b = 0$, using the different operations offered by Mermera. The program solves a system of randomly generated equations in 1000 variables. To ensure convergence of the asynchronous iteration, 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. 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

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: 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 Figure 8. A purely sequential Gauss-Seidel iteration that does not use Mermers converged in 143 seconds after doing 42 iterations.

time needed for the propagation of A and b to all processes.

A number of buffer sizes between 1 and 1000 writes are used to determine the effect of different buffer sizes on the performance of the solver. The performance of the program in Figure 8 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.

Measurements

Our measurements are summarized in Table 5. The fastest convergence times regardless of the buffer size used are shown. The measurements for each buffer size can be found in [29].

The main conclusions we derive from these data are:

1. Using noncoherent 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 multiple processes with noncoherent behavior yields significant speedup over the sequential case.
3. When we increase the number of processors from 1 to 2 and use noncoherent behaviors we observe a superlinear speedup — the performance improves by a factor of 3 even though the number of processors has only been doubled. This is because in the case of two processors working in parallel, each processor can use more recent values in each iteration than when only one processor is used. As the number of processors is increased the quadratic message complexity of the implementation comes into play resulting in congestion in the network and extra time spent by each process in receiving and sending messages. This explains the slower growth in the speedup as the number of processes is increased

Effect of buffer size: The effect of buffer size on the total time taken for the iterations of the solver to converge is shown in Figure 9. For any number of processors the convergence time is least

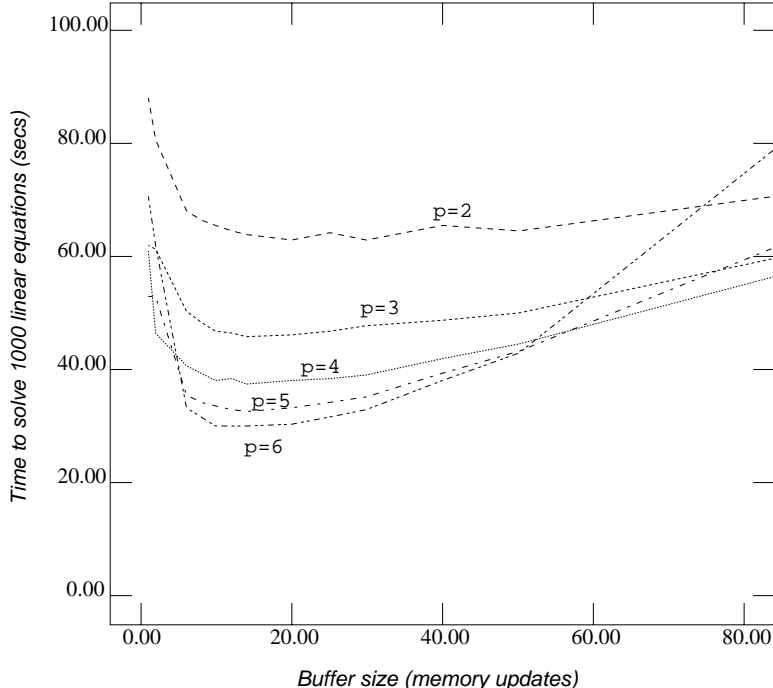


Figure 9: Execution time depends more acutely on buffer size as the number of processes p (i.e., degree of parallelism) is increased.

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 and can also result in congestion in the network. If the buffer is large then the frequency of messages is low but each process uses less recent values of the components of x being computed by other processes.

7 Related Work and Discussion

Ahamad *et al.* [3] present a formal model for describing causal memory, but they do not extend it to other noncoherent behaviors, nor to the mixing of different coherence levels. Their proofs of correctness of data-race-free and concurrent-write free programs on causal memory are applicable to our model.

Attiya *et al.* [5, 6] formally define a mixed-coherence memory. They allow for *strong* and *weak* operations. An execution consisting of *strong* operations only, or *strong reads* and *weak writes* only, or *weak reads* and *strong writes* only, is coherent. An execution of *weak* operations only is similar to the one permitted by our definition of *slow* memory, except that *weak* operations must be reliable. The set of executions allowed by their mixed-coherence memory lies between that permitted by a mixture of *coherent* and *PRAM*, and the set allowed by a mixture of *coherent* and *slow* in our model. Therefore all programs that run correctly on their mixed-coherence memory will run correctly on a mixture of *coherent* and *PRAM* memory. The difference between our mixture of *coherent* and *slow* memories and their memory is that our model allows *slow* operations to be propagated in a “best-effort” manner, i.e., they may not be observed by other processes, whereas their weak operations need reliable message transmission.

Singh [28] defines *causal* and *PRAM* behaviors separately, using a refined notion of commutativ-

ity [31]. He develops conditions under which such executions are *sequentially consistent*. Agrawal *et al.* [2] give a formal definition for a mixture of *causal* and *PRAM* memories, in which they label reads, but not writes. They define a *PRAM* history in terms of the causality of subhistories, as we do. The most salient difference between their approach and ours arises because they do not model the interactions between the *PRAM* portion of a history and the *causal* part. They neither require the *PRAM* subhistory to respect the fact that the *causal* subhistory is strictly stronger—and hence must be a subset of the *PRAM* one—nor do they have the *causal* subhistory recognize information flowing between its events through the *PRAM* subhistory, as we do. The three examples offered in [2] include a linear equation solver that differs from our version, by being both *synchronous* and centralized, which is unnecessarily restrictive.

With respect to our performance data, one may argue that this comparison of the completion time of coherent writes and noncoherent writes is unfair because few coherent memories are implemented using full replication. More efficient implementations such as directory based schemes (see [14] for an example) exist. Our response to this argument is that there are some applications (*e.g.*, the linear solver of Section 5) which intrinsically generate a message traffic that 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.

Our results suggest that network congestion can become a factor in applications using noncoherent memory. This is because noncoherent memories allow implementations in which the latency of an operation is independent of the message latency of the network. Thus applications can flood the network with messages causing message loss and consequent congestion. In contrast, the latency of coherent operations is dependent on the message latency of the network. Therefore, applications using coherent memory are self-limiting in their message generation rate. The impact of network congestion on the performance of asynchronous iterative methods that use noncoherent memories is studied further in [18, 19].

Our measurements of the performance of the equation solver reveal that asynchronous iterative methods can exhibit superlinear speedup when executed on noncoherent memory. This speedup occurs because the parallel asynchronous method's update of x_i may use more recent values of the other elements of x than are available in a sequential implementation. It is for the same reason that a Gauss-Seidel iteration converges faster than a Jacobi iteration, for the former allows permits new values computed in the current iteration to be used as soon as they become available. By contrast a Jacobi iteration insists that only values computed in the previous iteration influence those updated in the current one. However, the superlinear speedup is not sustained as the number of processes increases because of the overhead of sending and receiving more messages.

Prototypes of Mermera have been implemented on a network of workstations, a CM-5 and a BBN Butterfly TC2000. The implementation of *CO_Write* on the BBN Butterfly described in [29] uses a highly optimized pipelined locking protocol. The raw memory performance on these platforms corroborate the potential for significant performance improvements in some applications.

8 Conclusion

In this paper we presented a formal model for describing the behavior of several proposals for noncoherent memories. Using this formalism we proved that asynchronous iterations converge even when they execute on *slow* memory with a liveness guarantee. However, *detecting* the convergence requires a stronger memory which brings out the need for the ability to have different types of behaviors in the same program. Our formalism describes a correctness condition for programs

with mixed behavior. This paper also described Mermera, a system that provides the programmer with three types of noncoherent behaviors and one coherent behavior based on the formal model developed earlier in the paper. In this paper, the performance of a linear equation solver using the different behaviors provided by Mermera on a network of workstations is compared with that of a program that uses only its coherent behavior. This comparison, despite being within the confines of Mermera, suggests that dramatic performance improvements can be realized for some applications by using noncoherent memory instead of coherent memory.

Acknowledgements. Trung Dung supplied an early version of the if part of the proof of theorem 1. We would like to thank Hewlett-Packard's Metrix Network Systems, Inc., for allowing us free use of their network monitoring software.

References

- [1] 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.
- [2] D. Agrawal, M. Choy, H.V. Leong, and A.K. Singh. Mixed consistency: a model for parallel programming. In *Proc. 13th ACM Symp. on Principles of Distributed Computing, Los Angeles, California*, pages 101–110, Aug. 1994.
- [3] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
- [4] Mustaque Ahamad, Philip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. Technical Report GIT-CC-90-49, Georgia Institute of Technology, College of Computing, 1990.
- [5] 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.
- [6] Hagit Attiya, Roy Friedman, Soma Chaudhuri, and Jennifer L. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures, Velen, Germany*, pages 241–250, June 1993.
- [7] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, April 1978.
- [8] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1992. Ch 3 surveys queueing theory; ch 2 describes ISDN, TCP/IP, etc.
- [9] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [10] 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.

- [11] K. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. 11th ACM Symp. on Operating System Principles, Austin, Texas*, pages 123–138, Nov. 1987.
- [12] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, Aug. 1991.
- [13] John B. Carter, John K. Bennett, and Willy Zwaenopel. Implementation and performance of Munin. In *Proc. 13th ACM Symp. on Operating System Principles, Asilomar, California*, pages 152–164, Oct. 1991.
- [14] David Chaiken, John Kubiawics, 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.
- [15] Michel Dubois. Delayed consistency protocols. Technical Report CENG 90-21, Electrical Engineering–Systems Department, University of Southern California, July 1990.
- [16] Kourosh Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990.
- [17] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin-Madison, February 1991.
- [18] A. Heddaya and K. Park. Mapping parallel iterative algorithms onto workstation networks. In *Proc. 3rd IEEE International Symposium on High Performance Distributed Computing, San Francisco*, pages 211–218, Aug. 1994.
- [19] A. Heddaya, K. Park, and H.S. Sinha. Using warp to control network contention in Mermera. In *Proc. 27th Hawaii International Conference on System Sciences, Maui, Hawaii*, pages 96–105, Jan. 1994.
- [20] A. Heddaya and H.S. Sinha. Coherence, non-coherence and Local Consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Boston Univ., Computer Science Dept., May 1992.
- [21] A. Heddaya and H.S. Sinha. Computing with non-coherent shared memory. Technical Report BU-CS-93-007, Boston Univ., Computer Science Dept., Feb. 1993.
- [22] A. Heddaya and H.S. Sinha. An overview of MERMERA: a system and formalism for non-coherent distributed parallel memory. In *Proc. 26th Hawaii International Conference on System Sciences, Maui, Hawaii*, pages 164–173, Jan. 5–8 1993.
- [23] 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.
- [24] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.
- [25] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sep. 1979.

- [26] 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.
- [27] A.K. Singh. A framework for programming using non-atomic variables. In *Proc. 8th IEEE International Parallel Processing Symposium*, pages 133–140. IEEE Computer Society Press, 1994.
- [28] A.K. Singh. A framework for programming using non-atomic variables. In *Proc. 8th IEEE Intl. Parallel Processing Symp., Cancun, Mexico*, pages 133–140, Apr. 1994.
- [29] H.S. Sinha. *MERMERA: Non-coherent Distributed Shared Memory for Parallel Computing*. PhD thesis, Boston University, (<http://www.cs.bu.edu/techreports>), May 1993.
- [30] P. Wang and W.E. Weihl. Scalable concurrent B-trees using multi-version memory. *J. Parallel and Distributed Computing*, 32(1):28–48, Jan. 1996.
- [31] William E. Weihl. The impact of recovery on concurrency control. In *Proc. 8th ACM Symp. on the Principles of Database Systems, Philadelphia, Pennsylvania*, pages 259–269, Mar. 1989.