Princeton University

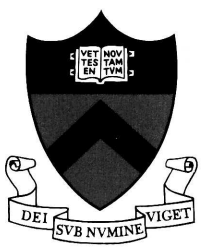
PRAM: A SCALABLE SHARED MEMORY

Richard J. Lipton
Jonathan S. Sandberg

CS-TR-180-88

September 1988

Department of Computer Science



PRAM: A Scalable Shared Memory

Richard J. Lipton
Jonathan S. Sandberg

Department of Computer Science
Princeton University

1. Introduction

Shared memory is studied and used extensively throughout the fields of VLSI design [26][27][28], MIMD architectures [10][25][35][50][51], parallel programming [39], distributed computation [42], and computer networking [14][37]. Shared memory systems are preferred to message passing schemes [30][45] because they are easy to program while at the same time they deliver high performance at a modest cost. The greatest weakness of shared memory systems is that they are not scalable systems. In other words, at some point the addition of hardware resources to a shared memory system degrades rather than improves system performance. Although there are dozens of shared memory MIMD machines commercially available, typically only a few processors can efficiently share memory. In MIMD architectures the latency of shared memory is one of the fundamental parameters determining the ultimate operation and performance of the entire system [53]. Hence there is intense interest throughout the commercial, military, scientific, and academic communities in the design, implementation, and analysis of shared memory systems with special attention to the latency of such systems.

Nearly all of the current shared memory designs implement coherent memory systems (*CRAM* systems). Such coherent systems have the property that a read of a memory address always returns the latest value written to that memory address. *CRAM* is desirable since it is a simple memory model that is both familiar and transparent to the programmer. A programmer can use a *CRAM* as if it were a single global memory. One common implementation of a *CRAM* shared memory system associates a local memory with each processor. Since the local memories may store multiple values associated with a single memory address, all *CRAM* systems include some mechanism or protocol to insure that local memories behave consistently.

The cost of the simplicity, familiarity, and transparency of *CRAM* is hardware complexity leading to increased expense. Current shared memory read and write commands require slow global actions that need access to a bus or a network resource. Some *CRAM* protocols require such global actions during the execution of each shared memory write command [2]. In the design of MIMD architecture caches, consistency requires a large processor state which in turn implies expensive context switching. The consequences of coherency are inefficient processor utilization during memory accesses and increased hardware expense. The search for a high performance *CRAM* protocol that requires minimum global action and minimum processor state is generally considered to be the outstanding open problem in computer architecture [53][6][7].

Recent investigations have demonstrated that it is possible to reduce the penalty of high interconnect latency costs at the expense of maintaining memory coherency. The goal of these investigations is to avoid processor idle time due to interconnect latency while maintaining low synchronization costs. In the *Horizon Project* [35] Smith has designed a machine that can effectively read and write in time independent of the interconnect latency. Valiant [41] proved a simulation theorem using the same idea to get an optimal simulation result. In both investigations memories were allowed to become inconsistent during the execution of independent streams of instructions. Since reads and writes from independent streams do not interact, the step by step consistency of the memory is not required. Scheurich and Dubois [44] have developed a cache system that sacrifices cache coherency for a weaker property: sequential consistency. Given the trace driven simulation results of Eggers [21], which indicate that the amount of write sharing in parallel

programs is small, shared memory systems that optimize performance with respect to memory latency may achieve much higher performance than other shared memory systems.

The major result presented in this paper is an analysis of a new scalable shared memory system called *PRAM* [37]. *PRAM* is scalable, unlike any *CRAM* shared memory system, precisely because *PRAM* shared memory can reach an inconsistent state. The effects of any memory inconsistency demonstrated by *PRAM* can be simply avoided using standard programming techniques. In fact, a large class of *CRAM* programs execute to completion exactly the same way on a *PRAM* as on a *CRAM*. Our approach to shared memory is similar in spirit to the MIPS project [26][27] approach to processor design. MIPS demonstrated that removing hardware locks on the instruction pipeline allows processors to run faster, while *PRAM* demonstrates that removing coherency requirements on shared memory systems increases the memory bandwidth. In both cases compiler technology is used to construct fast correct programs. In the following sections we will provide evidence that *PRAM* systems are simpler, faster, and less expensive than *CRAM* systems.

2. CRAM

In order to obtain results that are broadly applicable, we will use a very weak definition of a *CRAM* system, namely: a shared memory is a *CRAM* provided the results of any parallel computation is always that of *some* serialization. That is, every parallel computation reaches a state that some sequential computation could have reached. The usual notion of coherency, that each read returns the value of the last write, is a much stronger condition [17][44]. The weak consistency property will be examined under the condition that signal propagation delay or the network latency is not negligible.

A key parameter in any shared memory system is the *latency* τ. In a MIMD system the time required for signals to travel from one processor to another is called the system's latency. More generally, latency is the delay between a request or command and its corresponding fulfillment or completion. For the purposes of analysis, τ can be thought of as the sum of two components,

$$\tau = \tau_{global} + \tau_{local}.$$

τ_{global} is that portion of the latency delay that can be attributed to the execution of some global action such as requesting and receiving information over a common bus or interconnection network. τ_{local} is that portion of the latency delay that can be attributed to the execution of some local action. The cycle time of the local processor or the latency of a local memory access are examples of τ_{local} delays. If an atomic local action such as a write to local shared memory requires an accompanying action such as an acknowledgement from a remote memory, the τ_{global} component of the latency may be the dominating term in τ. Since any scalable shared memory system's performance must be immune to τ_{global} growing large with respect to τ_{local}, our analysis will focus on these two quantities.

There are two basic lower bounds on τ:

Physical: $\tau \geq d/c$ where d is the distance between the two processors and c is the speed of light; and

Logical: $\tau \geq \rho l$ where ρ is the gate delay of a basic gate and l is the minimum number of gates on any logical path from one processor to another.

The physical and logical lower bounds represent the two basic ways in which latency becomes a rate determining factor in data communications. Both lower bounds account for instances where τ_{global} dominates τ_{local} in the latency associated with a shared memory request. The physical inequality nearly becomes tight when the interconnect latency is attributed to the long distance that the signal needs to travel. Such would be the case in a wide area computer network connecting high performance CPUs over hundreds or even thousands of meters. But the physical bound, d/c, is even important in cases that signals travel only a few meters, as can be attested to by designers of supercomputer class multiprocessors. As processor speeds increase the physical latency bound will become as important in the fields of MIMD architecture and VLSI design as it already is in the fields of distributed computing and computer networking.

The logical inequality presented above is usually stronger than the physical inequality. The types of latency, τ, for which the logical bound dominates are network latency delays. These are typically associated with complex routing networks that provide flexible communication between a possibly large number of processors. For example, on a network based multi-processor such as the BBN Butterfly [13], ρl can be as large as several microseconds.

The following simple theorem is quite useful because it demonstrates that *CRAMs* must couple global actions (such as acknowledgement signals from remote machines) with local actions (such as reading and writing from local shared memory). In the case that τ_{global} dominates τ_{local}, *CRAM* commands must pay τ_{global} latency delays on local memory accesses.

Theorem 1: Let r (resp. w) be the best case (fastest possible) time to read (resp. write) in some *CRAM*. Then, $r+w \geq \tau$.

Proof. Let the *CRAM* have two processors named P and Q. Let x and y be shared variables which are initially both 0. Then let the processors execute the following programs:

P:
write (x,1);
w← read (y);

Q:
write (y,1);
$z \leftarrow$ read (x);

Clearly, in any serialization of these two programs (w,z) is equal to (0,1) or (1,0) or (1,1). It is never equal to (0,0) in any serial computation.

Since we have assumed that x and y are stored in a *CRAM*, any parallel computation must also avoid the state of (w,z) equal to (0,0). Now assume that both P and Q execute their statements at the same time, also assume that $r+w$ is less than τ. Then both w and z must be 0 following the completion of P and Q. Neither P nor Q can detect that the other has written because τ is the minimum information transfer time. This contradiction proves the theorem. QED.

Theorem 1 shows that either $r \geq \tau/2$ or $w \geq \tau/2$. We conjecture that a more delicate argument can improve that constant of 1/2 in these inequalities.

This simple theorem is important since it shows that the cost of consistency is that read/write latency must grow at least as fast as τ. Thus, no matter how clever or complex a protocol is, if it implements a coherent shared memory or *CRAM* , it must be "slow." If a shared memory system must be consistent, then it must take time proportional to τ_{global} for reading and writing. In particular, a consistent shared memory cannot be both fast (memory access time independent of τ_{global}) and scalable (allow $\tau_{global} \gg \tau_{local}$). In the next section we will define a *PRAM* shared memory that is allowed to become inconsistent so that it can read and write faster than τ_{global}.

All shared memory system implementations fall between two points in the design space. The first point has all p processors sharing a single physical memory. In such a shared memory the fundamental problem is that at least $p-1$ of the processors are physically separated from the shared memory. Thus, the time required by these processors to perform a read or a write is dependent on τ. As τ_{global} grows large with respect to the the processor speed the memory access latency becomes unacceptably long. Hence, few current designs favor this portion of the design space [14].

The second design point is a distributed implementation of p processors each with a local physical memory that communicate with a protocol that provides a virtual single shared memory. The fundamental problem facing this second implementation is that while reads and writes may be satisfied from local memory thereby avoiding latency delays, in the absence of further processing, individual constituent memories may respond differently to identical requests issued by distinct processors. The second implementation has been the focus of nearly all the published shared memory literature.

A simple coherent distributed implementation would not allow a write to a shared memory address to complete until all the other local memories containing that shared address were similarly written. In such a

scheme all the shared writes require time depending on τ because all the remote memories even if they do not contain the shared location must send an acknowledgement that they do not contain the location. The current battery of cache coherency protocols including: directory schemes [2][34], snoopy schemes [26], and software (compiler) enforced coherency schemes [10] [11] ameliorate the direct dependence of of write delays on τ demonstrated in the above simple example. In effect, these schemes seek to manipulate shared variables so that if a processor must write to a shared variable it nearly always writes to a local version of the variable and remains oblivious to the ensuing global actions.

The literature is filled with designs of coherent shared memories [2][4][6][9][17][24] [28][41][46][49]. Coherency is not so essential a property that all future shared memory designs need be coherent. Consider the two major benefits of coherent shared memory: it handles write-write conflicts efficiently and it provides fine grain locking. Eggers [20] has indicated that write contention is a small fraction (2%) of the total memory requests in common parallel programs. If this is true then a coherency protocol that significantly delays 98% of all memory references to aid the remaining 2% is of dubious value. More importantly, multiprocessor applications utilize multiple processors to update records stored in memory. Often these updates modify several words in the record. If the application is to keep these records in a consistent state usually only *one* update on each record can be allowed at a time. Thus, the applications must use a software *lock* to ensure that only one update occurs at a time. The coherency of the shared memory does not alleviate the need for these software locks. Applications using coherent memory, i.e., *CRAM*, or an incoherent memory such as *PRAM* both require synchronization overhead.

3. PRAM

In this section we present the definition of a *PRAM* (†) shared memory system and examine its main features. Let $P_1, P_2, \ldots, P_k$ be processors that share a memory with locations $0,1,\ldots,m-1$. Assume that each processor has a local memory M_i with memory locations $0,1,\ldots,m-1$. All the local memories are initially in the same state. The memory M_i is the i^{th} processor's view of the current state of the shared memory. Each processor executes read and write commands:

(1) *read*(i). Processor P_j executes this operation by performing a normal read from location i from its own local memory M_j.
(2) *write*(i,v). Processor P_j executes this operation by performing a *local* action and initializing a *global* action. Locally, it does a normal write to its memory M_j at location i with value v. Globally, it sends a message $<i,v>$ to all the other processors. No acknowledgement of successful message transmission is expected nor is one ever received.

As these messages $<i,v>$ arrive at a processor they are automatically executed (i.e., location i is written with value v). The data transmission media or switching network performs the task of repeating the message $<i,v>$ to all shared memory processors. In our implementations the *PRAM* memories have been dual ported (one port devoted to network messages and the other to the system bus). In both reading and writing, a processor never waits for the completion of a global action (in particular the execution time is independent of τ_{global}).

Theorem 1 shows immediately that a *PRAM* is not a coherent shared memory. The reader can readily observe that a *PRAM* system can execute a sequence of write commands that will cause the shared memory to become incoherent. This potential for inconsistency is the key feature of *PRAM*. The inconsistency alone is not a valuable feature, it is simply the necessary by-product of the complete decoupling of global actions and local shared memory accesses. The only global action initiated by the *PRAM* protocol has no completion acknowledgement mechanism. Hence, atomic local actions never wait for global actions to complete.

Since *PRAM* is not even sequentially consistent, no *CRAM* hardware implementation (i.e., Snoopy caches, Directory schemes, Distributed Map schemes, or weakly consistent schemes) can ever be behaviorally equivalent to a *PRAM* hardware implementation. In addition to behavioral inequality, *PRAM*

† Pipelined RAM

hardware is distinguished from *CRAM* with respect to financial and architectural costs. A *PRAM* implementation will always be less expensive than a comparable *CRAM* implementation because of economies in part count, printed circuit board complexity, manufacturing, and system test. The simplified *PRAM* architecture offers better use of technology related resources, faster design time, increased reliability, and migration of complexity to the compiler.

Although architectural simplicity and cost alone are sufficient to justify the adoption of new technology, *PRAM* also enjoys a performance advantage over *CRAM*. In computer architecture it has long been recognized that processors need not wait for each operation to complete before starting the next one. *PRAM* simply applies the same pipelining idea to shared memory block data transfers. The pipeline in *PRAM*'s case can be thought of as a signal transmission media with a long propagation delay. The most efficient way to use such a pipeline is to continuously load data signals into the transmission media. The strategy optimizes for speed regardless of how long the signals will be delayed before reaching their destination. For example, video signal transmission over satellite links commonly employ a block data pipeline.

PRAM achieves a performance advantage over *CRAM* in any shared memory application that requires a large fraction of local memory accesses to additionally perform some slower global action (i.e., when $\tau_{global} \gg \tau_{local}$). A *PRAM* system, provided with adequate switching connections, can scale to any number of nodes. There is no point where the signal propagation delays due to network saturation can cause the system to fail. Since the local *PRAM* is independent of τ_{global} this same quantity can grow very large without effecting the *PRAM* hardware.

4. PRAM Programs

Since *PRAM* and *CRAM* shared memory implementations are behaviorally distinct it cannot be the case that *PRAM* and *CRAM* programming are equivalent. Preliminary investigations indicate that *PRAM* programming is very simple, however, final judgement can only be made after the computer science community gains more extensive *PRAM* programming experience. The differences between *PRAM* and *CRAM* programming evolves solely from dependence on sequential consistency. We will examine these differences with respect to both coding practices (code generation) and execution speed. The unique *CRAM* codes that cannot be executed directly on a *PRAM* system can be identified by a compiler that produces correct *PRAM* versions of *CRAM* codes. Coding practices do not significantly change in moving code between systems. The execution speeds of many *CRAM* codes rise dramatically when transferred to a *PRAM* system.

Parallel programs perform three principal tasks: synchronization, data motion, and computation. *PRAM* software can perform all the functions programmers depend on in developing codes to perform these tasks. However, this is not the whole picture: in many cases the *exact same* program that solves the task on a *CRAM* will also operate correctly on *PRAM* system. This class of programs is important enough to name: say that a parallel program P is *conservative* provided it reaches the same final states on a *CRAM* as on a *PRAM*. The class of conservative programs is quite large, being composed of programs whose primary duty is data motion and computation.

The running time of *PRAM* programs will in general be as good or better that of corresponding *CRAM* programs. In order to make this comparison we will need the following definition. The number of *steps* of a parallel computation is the maximum number of operations performed by any of its processors. On the other hand, the *time* of a parallel computation is the actual duration of the computation taking into the account the execution times of each operation. Then,

Theorem 2: For any *CRAM* computation of n steps and t time, $O(\tau n) \geq t$. Moreover, $t \geq \Omega(\tau n)$ for some computations.

Proof: This follows directly from the definition of time and Theorem 1. QED

Theorem 2 shows that there exist a class of *CRAM* programs whose running time depends multiplicatively on τ. In sections 4.2 and 4.3 we will define and give examples of practically important classes of programs whose running time on a *PRAM* system depends only additively on τ. First we will examine the methods of synchronization available to *PRAM* and compare their performance with corresponding *CRAM*

synchronization methods.

4.1. Synchronization

Of all parallel programming tasks synchronization is the most difficult and hence requires the most attention in both *PRAM* programming and parallel computing in general. Synchronization is the only task where *CRAM* and *PRAM* programs display distinct behavior. Independent instruction streams performing data motion or computation execute the same way on a *PRAM* as on a *CRAM*.

Parallel programs use three basic synchronization mechanisms to: (1) synchronize the transfer of data between processes (producer-consumer), (2) hold processes at a common synchronization point (barrier), and (3) allow a process exclusive access to a system resource (mutual exclusion). Classic *CRAM* programs performing these functions have been published and thoroughly studied. The classic solutions use only read and write operations with the additional guarantee of serializability.

In order to demonstrate that *PRAM* can efficiently handle the common demands for synchronization we will show there are competitive *PRAM* programs for producer-consumer, barrier, and mutual exclusion. However, a great deal of research remains to be done on issues of fault-tolerance, fairness, and other aspects of synchronization on *PRAM*.

The *PRAM* programs we will propose as solutions to the producer-consumer and barrier synchronization tasks depend on the busy-wait loop. The standard busy-wait program where a process waits for a flag to change:

$flag \leftarrow 1$;
while ($flag = 1$) do;

is conservative, and hence works just the same on both *PRAM* and *CRAM*. In Theorem 3 we will use the busy-wait loop to implement conservative program solutions for producer-consumer and barrier.

Theorem 3: (1) Both *CRAM* and *PRAM* can solve the producer-consumer problem in the same time of $O(\tau)$ per item transferred. (2) Both *CRAM* and *PRAM* can perform a barrier in time $O(\tau log(p))$ with p processors.

Proof: We will just prove (2) since (1) follows in the same way. Both *CRAM* and *PRAM* use the same tournament method to generalize the solution for $p=2$ to $p>2$. For $p=2$ they each operate as follows: Each of the processors P and Q have flags: let P have x and Q have y as a flag. Initially, x and y are both 0. Then they execute the following programs:

P:
    $x \leftarrow 1$;
    while ($y=0$) do;

Q:
    $y \leftarrow 1$;
    while ($x=0$) do;

Clearly, this implements a barrier. Also it is easy to see that the time required by both *CRAM* and *PRAM* is $O(\tau)$. QED

Some of the classical programs used to solve the mutual exclusion problems on *CRAM* do not operate correctly on *PRAM*, i.e. they are not conservative. For example, Dekker's solution to the mutual exclusion problem [54] depends on the fact that the shared memory is serializable. Hence, it will not operate correctly on *PRAM*. However, there is a trivial *PRAM* program that does solve the mutual exclusion problem.

Theorem 4: Both *CRAM* and *PRAM* can solve the mutual exclusion problem in time $O(\tau)$ per operation.

Proof: We assign one of the processors to be the central controller. The other processor sends a message to the controller requesting access to the critical section. It then waits for the go ahead as in theorem 3. Clearly, the time bound is as claimed. QED

Theorems 3 and 4 demonstrate the existence of programming solutions to standard synchronization problems, but they do not necessarily indicate that *PRAM* is an easy target for which to write synchronization code. Current synchronization programs use a variety of primitives that take advantage of memory coherency. In general any commonly used synchronization primitive can be implemented in a *PRAM* program. Given a library of synchronization primitive routines it is easy for a compiler to substitute *PRAM* software for the *CRAM* operations. The result will be *PRAM* code that reaches the same set of final states as the original *CRAM* code.

4.2. Data Motion

PRAM executes data motion operations optimally. For any single data item *PRAM* and *CRAM* codes execute at approximately the same speed. However, for streams of data, *PRAM* programs take advantage of a pipeline effect. No matter how large τ_{global} becomes relative to τ_{local} data will be sent from the producer every τ_{local} time period until the data stream is exhausted. In Theorem 5 we will show that a *PRAM* can do a block copy of n words in time $O(n+\tau)$ rather than the expected *CRAM* time $O(\tau n)$. Thus, the latency τ, in a *PRAM* system, only contributes an additive term to the execution time. However, the latency τ execution time penalty in a *CRAM* system is expected to be multiplicative since latency costs may be paid at each step of the block copy. Many of the proposed shared memory systems would take $\Omega(\tau n)$ time; the few designs that avoid this do so only under special conditions using complex hardware.

Theorem 5: A *PRAM* can move n words from one memory buffer to another in time $O(n+\tau)$.

Proof: Let processor P wish to send n words to processor Q. Then P just copies the n words to Q's buffer. When this process is done, P signals the completion by setting a flag. Once the flag is set, Q is free to copy the block of words. The total time for this transfer is:

(1) $O(n)$ for P to copy the block of n words into *PRAM*;
(2) $O(\tau)$ for Q to get the result of setting the flag; and
(3) $O(n)$ for Q to copy the block from *PRAM*.

Thus the total time is $O(n+\tau)$. QED

The block transfer program used in the proof of Theorem 5 is another example of a conservative program. The same code executes correctly on the *CRAM* and *PRAM* but the *PRAM* code in this case requires the minimum possible execution time (†).

4.3. Computation

There are many examples of conservative computations because many parallel algorithms are organized into parallel stages that are executed in order. Within a stage each processor works independently on its own part of the computation. Typically, each processor operates in a manner that does not require that its writes be performed completely until the next stage is started. Since the processors are required in general to perform a barrier before the next stage it is always the case that the writes will be completed before the next stage starts. Thus, algorithms that are organized in this manner are always going to be conservative. These observations are summarized in the following theorem:

Theorem 6: Let $P_1 ; P_2 ; ...; P_k$ denote the sequential execution of the computations $P_1, P_2, ..., P_k$, while $P_1 \,||\, P_2 \,||\, \cdots \,||\, P_k$ denotes the parallel execution. Furthermore let $P_1, P_2, ..., P_k$ be conservative computations. Then,

(1) $P_1 ; P_2 ; ...; P_k$ is a conservative computation.
(2) $P_1 \,||\, P_2 \,||\, \cdots \,||\, P_k$ is a conservative computation provided no P_i writes a value that a P_j reads with $i \neq j$.

† Not accounting for encoding address bits

We have already seen an application of this theorem in the use of busy-wait loops for the construction of programs for producer-consumer and barrier in Theorem 3. Another important example is data base transaction processing. Transaction systems use software locks to control memory conflicts. Thus, they are conservative systems by Theorem 6.

We can greatly generalize Theorem 5 to the class of *oblivious computations*. A computation is oblivious if its data motion and the operations it executes at a given step of the computation are *independent* of the actual values of the data. Many important computations including the following are oblivious:

(1) Digital Signal Processing algorithms such as FFT [8];
(2) Matrix operations such as matrix-vector product, matrix-matrix product, and, elimination without pivoting; and
(3) Dynamic Programming [16].

Also included in the class of oblivious computations are systolic array algorithms [31], data-flow algorithms [2], and functional programs [25]. For example, most of the Livermore Loops [22] and LINPAC [15] codes are oblivious computations. The class of oblivious computations is very large.

On any computation let the *depth* of the computation be the minimum number of steps required to perform the computation with any number of processors. For example, FFT on n points has depth $log(n)$.

Theorem 7: Let an oblivious computation take n steps and have depth m. Then,

(1) A *CRAM* can do the computation with p processors in $O(\tau n/p+\tau m log(p))$ time; and
(2) A *PRAM* can do the computation with p processors in $O(n/p+\tau m log(p)$ time.

Proof: Since the computation is oblivious we can represent it by a fixed dag G. Each node of the graph corresponds to one step of the computation and an arc from one to another represents a dependency, i.e. $x \rightarrow y$ means that x must be executed before y. Both our claims (1) and (2) follow again from the same algorithm. Divide the graph G into its levels $G_0,G_1,...,G_{m-1}$ where G_i is the set of nodes with exactly i predecessors. Now execute each level in parallel and then do a barrier on all p processors before stating the next level. Clearly, this takes $|G_i|/p$ to do the i^{th} level and $O(\tau log(p))$ to do the barrier by Theorem 3. The total time is,

$$\sum_{i=0}^{m-1} \lambda \frac{|G_i|}{p} + O(\tau \log(p))$$

where λ is the cost of read/write access to the shared memory. Thus, the total time is

$$O(\frac{n\lambda}{p} + \tau m log(p)).$$

Setting $\lambda = \tau$ for a *CRAM and* $\lambda = 1$ for *PRAM* yields the theorem. QED

The importance of this theorem is that provided $n/p \gg m log(p)$ the first term dominates. In this case *CRAM* is about τ slower than *PRAM* in the worst case. Since this is often the case it follows that *PRAM* is much more efficient. Note, this is the case whenever the actual computation dominates the synchronization overhead. Also this theorem shows once again that both *CRAM* and *PRAM* often execute the *same* exact program. Only the time required is different.

5. Scalability

Corresponding to the two basic latency lower bounds presented in section 2 there are two fundamental directions along which one can scale a *PRAM* shared memory system. First, the distances between memories can be as long as desired. Second, the number of processors can be significantly increased. We will examine the effects of scaling *PRAM* along each of these directions.

Data communication is the key to successful exploitation of all types of parallel processing. *PRAM's* wide area high-speed memory-to-memory data transfer capability makes it uniquely suited for conducting

parallel computations across geographically isolated processors. Consider the execution time of a parallel program derived in Theorem 7,

$$O(\frac{n}{p} + \tau m \log(p)),$$

of n operations in m stages over p processors and a maximum latency τ. If we are willing to spend half our time performing interprocess communication then

$$\frac{n}{m} = \tau p \log(p)$$

In other words the amount of work per stage of the computation must be equal to the normalized latency delay ($\dagger$), times the number of processors, times the synchronization delay. If we are performing a calculation that requires 100,000 instruction executions per stage then in the case of four 1 MIP processors the normalized latency delay may approach 12,500. Such physical latency delays would not be encountered until the processors were approximately 2500 miles apart. If there is more work to be done per stage or if more time can be spent synchronizing the distances between processors can grow still larger.

Some computations may only be possible on the combined resources of geographically isolated machines. In these cases it is not the parallel speed up that is important but the fact that the task can be run at all. For example, consider scientific simulations/computations that use fine scales requiring hundreds of megabytes of memory for execution. Problems in aerodynamics, hydrodynamics, structural analysis, radiation transfer, thermodynamics, weather forecasting, solid-state physics, and physical chemistry all have such requirements. If no one machine has enough memory to run the simulation then the task cannot be completed. If the combined memory resources of four machines is available for one problem the simulation can proceed to completion.

PRAM can be scaled to support many more processors than conventional *CRAM*. However, there is one problem with a straightforward implementation of the *PRAM* protocol: network traffic. Consider a *PRAM* system with p processors. Then if each processor simultaneously writes to the shared memory, each processor must send p messages through the network; a total of p^2 messages will be sent. In a scalable system where p would be large this would be a major bottleneck. Thus, we must consider how to reduce the number of messages that the network must handle.

A simple solution to the network traffic problem is to add a cache to each processor. Then generalize the *PRAM* protocol to handle cache read and write misses.

(1) *read−miss*(i). *PRAM* behaves as any conventional cache would: a memory request is made for the specified word.

(2) *write−miss*(i,v). *PRAM* broadcasts a write message of the form $<i,v>$ to all the other processors. The write does not wait for any sort of acknowledgement.

The use of a cache has all the usual advantages and dramatically reduces network traffic. Assume that each cache has a table mapping pages between caches. Since messages, $<i,v>$, are only sent to those processors indicated in the page table the network traffic is reduced. For example, consider an arbitrary oblivious computation. If the fan-in of each operation is at most $O(1)$, then on the average the fan-out must be at most $O(1)$. Thus, each write message from a processor is needed by at most $O(1)$ other processors. The network traffic is reduced dramatically.

Existing codes have similar write fan-out upper bounds so that all caches need not be mapped into the same shared memory. However, even if one page is in all the caches only writes to it will cause messages to be sent to p processors. The network traffic will be modest provided there is not a substantial write-write sharing of data. Since the trace simulation results presented in [20] indicate that write sharing is rare it follows that *PRAM* is scalable with respect to the number of processors.

$\dagger$ The normalized latency is $\frac{\tau}{\tau_{local}}$

6. Conclusions

We have presented a new scalable implementation of shared memory, *PRAM*, that is immune to long network latency delays. Surprisingly, the cost of synchronization has remained low while the long latency performance has been significantly improved. The price for this scalable behavior is that the *PRAM* shared memory is not and cannot be coherent or even sequentially consistent. The possibility of a *PRAM* system becoming inconsistent is a simple problem to overcome with software. Moreover, a large class of programs, the class of conservative programs, behave exactly the same way on a conventional shared memory or a *PRAM*.

PRAM is currently running parallel processing codes and serving as the primary computer network in our laboratory at Princeton University. It is implemented on IBM PC/AT boards that contain 32 Kbytes of memory and fiber-optic connectors. Each board performs the *PRAM* protocol at full bus speeds; thus, each IBM PC "sees" 32KB of shared memory. Work on *PRAM* boards for VME and other busses is currently underway. All future *PRAM* boards will be compatible with previous *PRAM* hardware. Hence, the *PRAM* hardware will be used as a high-speed heterogeneous bridge. A variety of software projects are currently underway to exploit this system.

The unique features offered by *PRAM* including: scalability, low cost, simplicity, memory-bus speed data transfers, wide area distributed computing capabilities, and highly flexible resource sharing make *PRAM* an obvious candidate for the backbone of the Federal Coordinating Council on Science, Engineering and Technology (FCCSET) proposed National Computer Network. In the search for breakthroughs in parallel supercomputer simulation of physical processes modeled by nonlinear three-dimensional PDE's, *PRAM*-based caching strategies are worthy of inclusion in any MIMD supercomputer architecture. Finally, in VLSI MIMD architectures as increased processor speeds and silicon real estate create a demand for more on-chip cache *PRAM* will be an attractive method to lower the required off-chip bandwidth.

7. Acknowledgements

We would like to acknowledge and thank R. Altman, T. Altman, R. James, D. Serpanos, and C. Zimmerman for their collective efforts on behalf of the *PRAM* project. This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either express or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

8. References

[1] Agarwal, A., "Analysis of Cache Performance for Operating Systems and Multiprogramming," PhD. Th., Stanford, CSL-TR-87-332, 1987.

[2] Archibald, J. and Baer, J.L., "An Economical Solution to the Cache Coherence Problem," The 11th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 355-362, 1984.

[3] Archibald, J. and Baer, J.L., "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors," ACM Transactions on Computer Systems, Vol. 4, No. 4, pp. 273-298, Nov. 1986.

[4] Arvind and Iannucci, R.A., "A Critique of Multiprocessing von Neumann Style," The 10th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 426-436, 1983.

[5] Axelrod, T.S., "Effects of Synchronization Barriers on Multiprocessor Performance," Parallel Computing, Vol. 3, No. 2, May 1986.

[6] Bitar, P. amd Despain, A.M., "Multiprocessor Cache Synchronization" The 13th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 424-433, 1986.

[7] Bitar, P., "Fast Synchronization for Shared Memory Multiprocessors," RIACS TR 85.11, Dec. 1985.

[8] Brigham, E.O., The Fast Fourier Transform, Prentice-Hall, 1974.

[9] Cheong, H. and Viedenbaum, A.V., " A Cache Coherence Scheme with Fast Selective Invalidation," The 15th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 299-364, 1988.

[10] Cheriton, D.R., et al., " The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation," The 15th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 410-421, 1988.

[11] Cheriton, D.R., Slavenberg, G.A., and Boyle, P.D. "Software-Controlled Caches in the VMP Multiprocessor," The 12th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 366-374, 1986.

[12] Clark, D., "Cache Performance in the VAX 11/780," ACM Transactions on Computer Systems, Vol. 1, pp. 24-37, Feb. 1983.

[13] Crowther, W., et al., "The Butterfly Parallel Processor," IEEE Arch. Tech. Comm. Nesletter, pp. 18-45, Sept./Dec. 1985.

[14] Devlin, et al.,"Shared Memory Multiprocessor Computer System," U.S. Patent No. 4,212,057, July 8, 1980.

[15] Dongarra, J.J., et al., LINPACK User's Guide, SIAM, 1979.

[16] Dreyfus, S.E. and Law, A.M., The Art and Theory of Dynamic Programming, Academic Press, 1977.

[17] Dubois, M., Scheurich, C., and Briggs, F., "Memory Access Buffering in Multiprocessors," The 13th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 434-442, 1986.

[18] Dubois, M. and Briggs, F.A., "Effects of Cache Coherency in Multiprocessors," IEEE Transactions on Computers, Vol. c-31, No. 11, Nov. 1982.

[19] Dubois, M., Scheurich, C., and Briggs, F.A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," IEEE Computer, Vol 21, No. 2, pp. 9-21, Feb. 1988.

[20] Eggers, S.J. and Katz, R.H., "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluations," The 15th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 373-383, 1988.

[21] Eggers, S.J., "Simulation and Analysis of Data Sharing Support in Shared Memory Multiprocessors," PhD. Th., Berkeley, 1988.

[22] Feo, J.T., "An analysis of the Computational and Parallel Complexity of the Livermore Loops," Parallel Computing, Vol. 7, pp. 163-185, 1988.

[23] Goodman, J.R. and Woest, P.J., " The Wisconson Multicube: A New Large-Scale Cache-Coherent Multiprocessor," The 15th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 422-431, 1988.

[24] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," The 10th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 124-131, 1983.

[25] Gottlieb, A., et al., "The NYU Ultracomputer -- Designing an MIMD Shared-Memory Parallel Computer," IEEE Transactions on Computers, Vol. c-32, pp. 175-189, Feb. 1983.

[26] Hennessy, J.L., et al., "MIPS: A VLSI Processor Architecture," Technical Report No. 223, Computer Systems Laboratory, Stanford University, Nov. 1981.

[27] Hennessy, J.L., "VLSI Processor Architecture," IEEE Transactions on Computers, Vol. c-33, No. 12, Dec. 1984.

[28] Hill, M.D. and Smith, A.J., "Experimental Evaluation of On-Chip Microprocessor Cache Memories," The 10th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 158-166, 1984.

[29] Hill, M.D., "Aspects of Cahce Memory ans Instruction Buffer Performance," PhD. Th., Berkeley, 1987.

[30] Hillis, W.D., The Connection Machine, MIT Press, 1985.

[31] Hillis, W.D. and Steele, G.L., "Data Parallel Algorithms," Communications of the ACM, Vol. 29, pp. 1170-1183, Dec. 1986.

[32] Karlin, A.R. et al., "Competitive Snoopy Caching," Algorithmica, Vol. 3, pp. 79-119, 1988.

[33] Karp, R.M. and Ramachandran, V., "A Survey of Parallel Algorithms for Shared Memory Machines," UCB Report No. UCB/CSD 88/408, March 1988.

[34] Katz, R.H., et al., "Implementing a Cache Consistency Protocol," The 12th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 276-283, 1985.

[35] Kuehn, J.T. and Smith, B.J., "The Horizon Supercomputing System: Architecture and Software," personal comm. 1988.

[36] Kung, H.T., "Synchronized and asynchronous parallel algorithms for multiprocessors," Algorithms and Complexity, pp. 153-200, 1976.

[37] Lipton, R.J. and Sandberg, J.S., "Oblivious Memory Computer Networking," U.S. Patent Application, 1988.

[38] Moore, W., McCabe, A., and Urquhart, R. (eds.), Systolic Arrays, Adam Hilger, 1987.

[39] Perrott, R.H., Parallel Programming, Addison Wesley, 1987.

[40] Priess, B.R. and Hamacher, V.C., " A Cache-Based Message Passing Scheme for a Shared-Bus Multiprocessor," The 15th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 358-364, 1988.

[41] Przybylski, S., Horowitz, M., and Hennessy, J., "Performance Tradeoffs in Cache Design," The 15th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 290-298, 1988.

[42] Raynal, M., Networks and Distributed Computation, The MIT Press, 1988.

[43] Rudolph, L. and Segall, Z.,"Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," The 11th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 340-347, 1984.

[44] Scheurich, C. and Dubois, M., "Correct Memory Operation of Cache-Based Multiprocessors," The 14th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 234-243, 1987.

[45] Seitz, C.L., "The Cosmic Cube," Communications of the ACM, Vol. 28, pp. 22-33, 1985.

[46] Smith, A.J., "Cache Memories," ACM Computing Surveys, Vol. 14, No.3, pp. 473-530, September 1982.

[47] Smith, A.J., "Bibliography and Readings on CPU Cache Memories and Related Topics," Computer Architecture News, Vol. 14, No. 1, pp. 22-42, Jan. 1986.

[48] Smith, A.J., "Cache Emulation and the Impact of Workload Choice," The 12th Annual International Symposium on Computer Architecture Conference Proceedings, pp. 64-73, 1985.

[49] Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," IEEE Transactions on Computers, Vol. c-36, No. 9, pp. 1063-1075, Sept. 1987.

[50] Swan, R.J., et al., "Cm* - A Modular, Multi-Microprocessor," Proceedings of the National Computer Conference, pp. 39-46, 1977.

[51] Thacker, C.P., Stewart, L.C., and Satterthwaite, E.H., "Firefly: A Multiprocessor Workstation," IEEE Transactions on Computers, Vol. 37, No. 8, pp. 909-920, August 1988.

[52] Valiant, L.G., Optimally Universal Parallel Computers, Unpublished Manuscript, 1988.

[53] Arvind and Iannucci, R.A., "Two Fundamental Issues in Multiprocessing," MIT Computation Structures Group Memo 226-5, July 25, 1986, pp. i-35.

[54] Dijkstra, E.W., "Co-operating Sequential Processes," Programming Languages, ed. Genuys, Academic Press, 1968, pp. 43-112.