A Study on the Impact of Memory Consistency Models on Parallel Algorithms for Shared-Memory Multiprocessors

Guojing Cong
IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598

October 6, 2005

Abstract

Memory consistency model is an integral part of the shared-memory multiprocessor system, and directly affects the performance. Most current multiprocessors adopt relaxed consistency models in quest of higher performance. In this paper we study the impact of memory consistency model on the design, implementation and performance of parallel algorithms for graph problems that remain challenging due to poor locality. Our study reveals interesting findings through a unified algorithmic and architectural approach. We show that relaxed models do not necessarily complicate algorithm design, and the simplicity and intuitive ordering constraints of sequential consistency can help reduce synchronization and achieve superior performance over algorithms for relaxed models. The graph algorithms we study constitute a new benchmark suite for evaluating the consistency model and other aspects of shared-memory systems. Our benchmark suite represents combinatoric workload and lays higher stress on the memory sub-system, and hence complements the SPLASH/SPLAH-2 suites for shared-memory multiprocessors.

Keywords: Memory Consistency Models, Parallel Algorithms, Shared Memory, High-Performance Algorithm Engineering.

1 Introduction

Shared-memory multiprocessors are widely used in parallel computing to solve challenging irregular problems as they provide a single-address space programming model that simplifies data partitioning and load balancing. As with uniprocessors, the memory sub-system design is critical to the performance of the system, and caches are widely used to hide long latency to main memory. There is extra complexity with shared-memory multiprocessors in maintaining coherence among caches and preserving ordering constraints to memory accesses from multiple processors. Memory consistency model (or memory model) defines a clear interface between processor and memory by specifying the order in which memory operations from multiple processors perform.

Intuitively, a memory read should return the last value written to the corresponding location, and a memory write should become instantaneously visible to all processors. Unfortunately such expectations fail to produce a well-defined ordering due to physical limitations. Sequential consistency (*SC*) is arguably the most widely understood memory model, and is oftentimes assumed as *the* model that governs the ordering of memory accesses. Under *SC* the system behaves logically as a multi-programmed uniprocessor, and memory accesses to shared memory are observed in the same order on each processor. The ordering constraint imposed by *SC* is restrictive for hardware implementation as memory accesses are serialized. In quest

of high performance, most current shared-memory multiprocessor designs adopt relaxed orderings (hence the name relaxed models) that allow more aggressive hardware optimizations. Relaxed models define only partial orderings on memory accesses. Accesses that are not ordered under the partial ordering are concurrent, and may be scheduled in arbitrary order to keep the processor busy. The complication is that different processor may observe different order of the same set of memory operations, and the same program may produce different results under different models.

Memory model has been studied extensively by the architecture community. Research on memory models has been focusing on the proposal of new relaxed models (e.g., see[1, 9]), performance comparison of different memory models (e.g., see [11, 3]), hardware optimizations that improve the performance of memory models (e.g., see [12, 26, 3, 13]), and compiler and language design for relaxed models (e.g., see [10]). The current consensus is that: relaxed models enable more aggressive hardware optimizations than strict models, and deliver better performance; the disadvantage of relaxed models, however, is that they complicate algorithm designs as they are considerably more involved to reason with.

In this paper we study the design and implementation of parallel algorithms on both sequentially consistent and relaxed models. Prior studies on memory models and parallel algorithms are largely independent. Memory models are transparent to theoretic algorithm designs as they are not reflected in the parallel models, while experimental studies run mostly scientific computing benchmarks that are relatively easy to parallelize. We consider parallel algorithms for graph problems. The irregular, sparse instances remain challenging even on shared-memory architectures due to poor locality features. Parallel algorithms for graph problems lay greater stress on the memory sub-system than scientific applications, and the design and implementation is more sensitive to the choice of memory models.

Our study reveals interesting findings for both parallel algorithm and memory consistency model research. We show that ordering constraints of memory models can be drawn on to design asynchronous algorithms with very few explicit synchronizations. We present an example – parallel spanning tree, where an asynchronous algorithm designed for *SC* achieves superior performance over algorithms for relaxed models. To our knowledge, this is the first non-trivial counter-example to the general belief that strict ordering brings poor performance. Our study with PRAM algorithms suggests that relaxed models do not necessarily complicate algorithm design. Prior belief (that they do) generally assumes some asynchronous models that evolve from the view of multiprogrammed uniprocessors. Our results signify the importance of algorithm study in architecture design, especially when parallelism is becoming the major source of performance improvement for future computers.

The rest of the paper is organized as follows. Section 2 gives a brief review of major classes of memory models. Section 3 discusses the impact of memory models on the design and implementation of PRAM algorithms, and compares the performance of PRAM algorithms on different memory models for several graph problems. Section 4 shows how ordering constraints of memory models can be used to design asynchronous algorithms. In section 5 we present our conclusions and future work.

2 Memory Consistency Models

SC was proposed by Lamport [21]. A multiprocessor is sequentially consistent if *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*. *SC* requires writes propagate automatically, and is restrictive for hardware implementation.

After *SC*, a series of relaxed models were proposed to enable aggressive hardware implementations by relaxing the ordering constraints. According to which constraints are relaxed, the models can be categorized

into two classes: "relaxing write-to-read program order" and "relaxing all orders". They are represented by processor consistency (*PC*) and release consistency (*RC*), respectively. Memory models of commercial parallel computers usually fall into the two categories, e.g., *IBM 370*, *Intel Pentium Pro*, and *Sun total store order (TSO)* relax write-to-read program order, while *DEC Alpha*, *IBM PowerPC* and *Sun relaxed memory order (RMO)* relax all program orders. A good survey of shared memory consistency models can be found in [2].

As memory models directly affect the performance of multiprocessors, there are studies that aim to improve the performance of memory models. Gharachorloo *et al.* [12] present two techniques, *hardware-controlled non-binding prefetching* and *speculative execution* that are shown to provide higher performance for all memory models. Hardware prefetching pipelines long latency memory accesses that would otherwise be delayed due to consistency constraints. Speculative execution allows out of order execution of load and store operations. The two techniques work especially well for *SC* and greatly reduce the gap of the performance between *SC* and relaxed models. In fact Hill argues that the performance gap is small enough that programmers should not be bothered with the extra burden of reasoning with relaxed models [16].

3 Impact of Memory Models on PRAM Algorithms

Algorithm designs that aim to expose the maximum parallelism of a problem oftentimes assume the PRAM model as it presents a very abstract view of parallel computers. Memory models are not reflected in PRAM. In fact, formal formulation of relaxed models emerge after the proposal of PRAM. It is important to study the impact of memory models on PRAM algorithms as there is a rich collection of PRAM algorithms in the literature.

PRAM is a lock-step synchronous model with a global clock. There is no cache in PRAM, and memory accesses in one step are performed before the next. A total ordering (which may not be unique) of the memory accesses can easily be formulated. Obviously *SC* is observed with PRAM. The memory model implied by PRAM is actually even stricter than *SC* as there is a global clock. We next discuss the implications of implementing PRAM algorithms on asynchronous architectures with sequential consistency and relaxed memory models.

3.1 Implementation of PRAM Algorithms on *SC* and *RC*

When emulating PRAM algorithms on current shared-memory multiprocessors, barriers are to be inserted to guard against races among processors. Assuming a strong consistency model, e.g., reads and writes are atomic and program order is preserved, it is straightforward to insert barriers at proper locations. Here barriers are introduced due to the asynchronous nature of the multiprocessors. A natural question that arises is whether more barriers or other forms of synchronization are necessary when there are caches and the memory model can be relaxed.

The synchronous nature of PRAM obviate the use of busy-waiting (e.g. flag variables and spinlocks) or mutual exclusion constructs (e.g., mutex locks) for coordinating processors in the algorithm design. We focus on PRAM algorithms without such asynchronous constructs for processor coordination, and argue that doing so does not restrict the range of algorithms that we consider. First, PRAM does not provide explicit support for locks or other mutual exclusion constructs. Second, the implementation of locks using only memory reads/writes (e.g., Lamport's bakery algorithm [20]) and the use of flag variables inevitably involves busy-waiting in an asynchronous setting. Such synchronization can be easily achieved by resorting to the synchronous nature of PRAM. Take Fig. 3.1 as an example. On the left are two code segments executing on

processors P_1 and P_2, respectively. The coordination between P_1 and P_2 are through flag variables. A natural way to synchronize on PRAM is shown on the right. Third, there is no race condition in accessing shared memory with EREW and CREW PRAM algorithms, and for CRCW PRAM the conflicts among processors are usually handled by the algorithm.

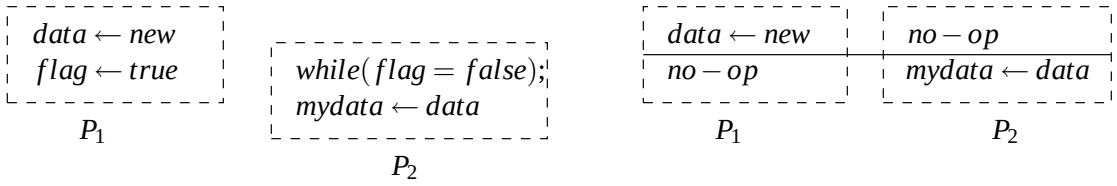

Figure 1: On the left: two processors coordinate using flag variable. On the right: the same coordination under a synchronous model

We first consider running PRAM algorithms on asynchronous PRAM (APRAM) with a strong consistency model. APRAM was proposed to account for the fact parallel computers are mostly asynchronous and synchronization cost is not to be ignored [7]. Assuming no asynchronous construct for coordination in PRAM algorithms, we consider the number of barriers needed to run an algorithm on current memory models.

Lemma 1 *Suppose k barriers are inserted to run a PRAM program $\mathcal{P}$ on APRAM, and no asynchronous coordination construct is used in $\mathcal{P}$. Then there is at least one barrier between any data dependency that involves two processors.*

Proof:

The k barriers $b_1, b_2, \ldots, b_k$ divide $\mathcal{P}$ into $k+1$ block $B_0, B_1, \ldots, B_k$. We show that for any B_i ($0 \leq i \leq k$), there is no data dependency between statements that run on different processors. Suppose on the contrary, inside some block B_i there exits data dependency between statement $s \in B_i$ on processor P and statement $s' \in B_i$ on processor P', $P \neq P'$. W.l.o.g., assume that logically s precedes s'. Since there is no barrier between s and s', even in case that processor P' proceeds at a much higher speed than P, it is still guaranteed that s' executes after s. Then there must be a busy wait loop $l \in B_i$ preceding s' on P' whose exit can only be triggered after the completion of s. Either that s and s' are part of synchronization using flag variables, or there is a flag variable synchronization construct sandwiched between them in B_i, a contradiction to our general assumption. As there is no data dependency between two processors inside a block and there is a barrier between two consecutive blocks, any such data dependency is separated by a barrier synchronization. □

For relaxed models, special ordering instructions can be combined with a barrier synchronization that guarantees the completion of any memory access reference before the synchronization point. We establish a theorem regarding the number of barriers needed for running PRAM algorithms on current memory models.

Theorem 1 *Suppose a minimum of k barriers are inserted to run a PRAM program $\mathcal{P}$ on APRAM. If the underlying memory model is changed to sequential consistency or relaxed models, a minimum of k barriers can be used to guarantee the correct execution of $\mathcal{P}$ if memory operations before a barrier are performed with regard to memory operations after a barrier.*

Proof: Suppose k' is the minimum number of barriers needed for sequential consistency or relaxed consistency. $k' \geq k$ as we can always add extra constraints into the model and make it behave like APRAM. On

the other hand, since there is a barrier between any data dependency between two processors, the k barriers work for relaxed models, hence $k \geq k'$. Now we have $k = k'$. □

Theorem 1 shows that with the inherent necessity of barriers to run PRAM algorithms on asynchronous architectures, the underlying memory model of the architecture can be relaxed if special memory ordering operations are combined with the barrier that guarantees the completion of memory operations before the barrier.

Algorithm design for PRAM and emulation on asynchronous architectures are oblivious of the underlying memory models. Relaxed models do not complicate the algorithm design or implementation. In our study with graph problems our implementations of PRAM algorithms are the same for different memory models, except for the implementation of barriers. This argues strongly for relaxed memory models as they allow for more aggressive hardware optimizations.

3.2 Performance of PRAM algorithms on *SC* and *RC*

We next compare the performance of PRAM algorithms for several graph problems on two memory models, *SC* and *RC*. We ignore *PC* as it is in between *SC* and *RC* in terms of ordering constraints, and the performance on *RC* is usually higher than that on *PC*.

Prior studies run SPLASH/SPLASH-2, or similar benchmarks [11, 3]. Most of the benchmark applications have algorithm designs for asynchronous architectures, and use locks to synchronize multiple processors. These benchmarks exhibit reasonable locality features [29]. In our study, we solve graph problems with arbitrary, sparse instances, including spanning tree (ST), minimum spanning tree (MST), connected components (CC), list ranking (LR) and biconnected components (BiCC). Although list ranking does not solve a graph problem itself, it is frequently used in parallel graph algorithm designs, e.g., finding the size of the subtree rooted at each vertex using the Euler-tour technique. The graph problems are important themselves with a wide range of applications. They are also used as building blocks for more complicated problems, e.g., tri-connected components and planarity testing. For connected components and spanning tree, we implement the Shiloach-Vishkin connected components algorithm and a derived spanning tree algorithm that runs two-phase election to resolve races among processors [27, 5]. For MST, we implement the parallel Borůvka algorithm [6]. The Tarjan-Vishkin algorithm is implemented for the biconnected components problem [28, 8]. These algorithms employ many of the fundamental parallel primitives, e.g., prefix sum, pointer jumping, list ranking, sorting, and tree computations.

We run our implementations on the RSIM simulator that simulates modern processors and memory sub-system. Features of the processors include instruction-level parallelism, out-of-order scheduling, non-blocking reads and speculative execution. We use the MESI cache coherence protocol in the study. A cache size of 16KB is used, and the problem size is in the order of 64K. For random graphs, there are 64K nodes and 256K edges, and for lists, there are 64K elements. Note that the memory usage can be much greater than 64K as there are multiple fields of a data structure and there are also auxiliary data structures. The simulator simulates a 16-processor system, yet due to the very long simulation time, we run the algorithms with 2, 4 and 8 processors. We refer interested readers to [25, 3] for features of the simulator and parameters of the simulated architectures.

Compared with the SPLASH/SPLASH-2 benchmark suites, our graph problem benchmark lays higher stress on the memory sub-system. Table 1 compares the percentage of load and store instructions over the total number of instructions executed in both the SPLASH-2 benchmark and our graph problem benchmark. We take four programs from the SPLASH-2 benchmark (there are 12 programs in total, but some of them are similar) and calculate the percentage of loads and stores using simulation data from [29]. For our graph

problem benchmark, we collect the number of load, store and total instructions using hardware performance counters on IBM Power4 processors. We see there is a significantly higher percentage (on average, above 15% for the programs shown in the table) of load and store instructions in our graph benchmark than in the SPLAH-2 benchmark. We expect optimizations in memory sub-system have larger impact on the graph problem benchmark.

	SPLASH-2				Graph Problems			
Benchmark	Barnes	Cholesky	Ocean	Raytrace	ST	MST	BiCC	CC
load	20.3%	20.6%	21.4%	25.1%	45.6%	28.6%	40.4%	44.5%
store	15.6%	5.2%	17.8%	9.5%	2%	15.2%	19.2%	1.4%
load+store	35.9%	25.8%	39.2%	34.6%	47.6%	43.8%	59.6%	45.9%

Table 1: Comparison of the percentage of load and store instructions for the SPLASH-2 benchmark and the graph problem benchmark.

Fig. 2 shows the performance of the graph problem benchmark on *SC* and *RC*. On the left is the performance graph for base implementation of *SC* and *RC*. On the right is the performance graph for optimized implementation of *SC* and *RC* with hardware prefetching and speculative execution.

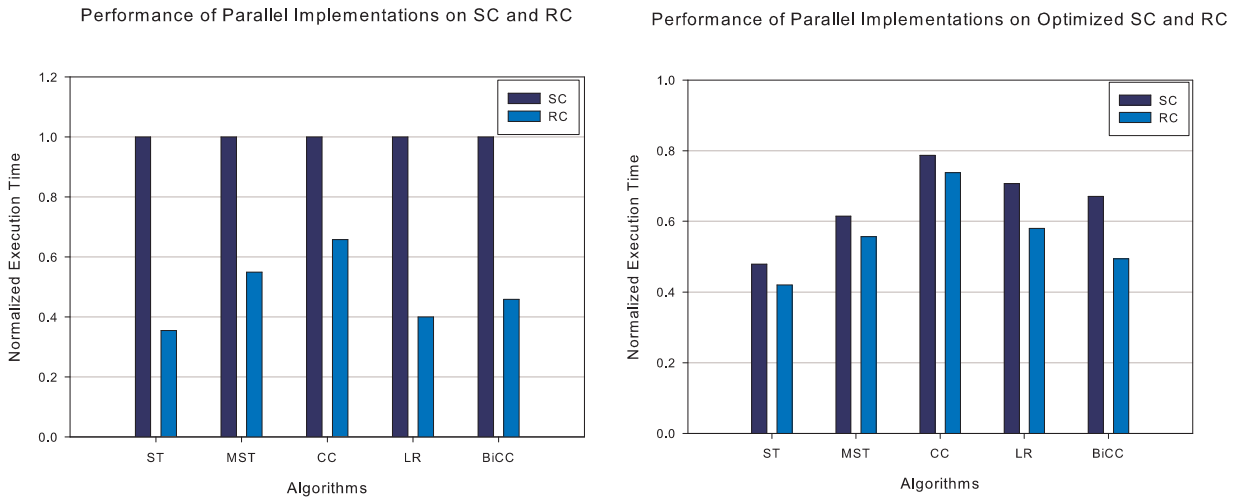

Figure 2: The performance of benchmark implementations on *SC* and *RC*. On the left is the performance for base implementations of *SC* and *RC*. On the right is the performance for optimized implementations of *SC* and *RC* with prefetching and speculative execution.

There is generally a big gap in performance for the parallel implementations on base *SC* and *RC*. Implementations on *RC* take roughly 40% to 60% (on average 48%) of the execution time of the corresponding implementations on *SC*. Performance improvement on *RC* over *SC* for the graph benchmark is higher than that of the SPLASH benchmark studied in [3] which shows implementations on *RC* take on average 54% of the execution time on *SC*.

Also we see that the two techniques proposed by Gharachorloo *et al.* greatly improve the performance of *SC* (approximately 40% to 60%). Yet the hardware optimizations do not show any significant improvement for the performance on *RC*. For biconnected components, even with optimized *SC*, the implementation is

still significantly slower than that on *RC* (more than 20% of reduction in execution time on *RC*).

4 Algorithms on *SC* that Reason with Ordering Constraints

We show in the previous section that PRAM algorithms are oblivious of memory models due to the synchronous nature of PRAM, and the implementations run faster on relaxed models. Yet we can not come to the conclusion that relaxed models should be adopted for parallel computing. In this section we take a look at the relationship between memory models and parallel algorithms from a different perspective.

PRAM algorithms do not always achieve good performance compared with the sequential implementations due to the parallel overhead, e.g., large algorithmic overhead, and significant communication and synchronization cost. In fact for several irregular problems straightforward implementations of PRAM algorithms do not even outperform the sequential implementation with a moderate number of processors (e.g., see [5, 6]). For shared-memory multiprocessors, the amount of parallelism offered by the PRAM algorithm greatly exceeds the number of available processors. Acknowledging the fact that only limited number of asynchronous processors are available, we design asynchronous algorithms for p processors where $p \ll n$ (n is the input size) that incur smaller algorithmic and synchronization overhead. Further more, synchronization cost can be reduced if the algorithm is able to coordinate processors by reasoning with ordering constraints on memory operations, which is especially advantageous for parallel algorithms with large irregular inputs that would otherwise require many synchronization operations.

We show as an example, a parallel spanning tree algorithm, that coordinates processors by reasoning with the ordering constraints of *SC*, and achieve superior performance on *SC* than PRAM algorithms on *RC*. There are several parallel spanning tree algorithms proposed in the literature (e.g., see [15, 18, 27]), yet many of them are impractical. The Shiloach-Vishkin parallel spanning tree algorithm (denoted as *SPAN_SV* in this paper) [27, 28] is representative of several connectivity algorithms that adapt the graft-and-shortcut approach, and is implemented and optimized in prior experimental studies (e.g., see [14, 17, 19]). For graph $G = (V,E)$ with $|V| = n$ and $|E| = m$, the algorithm achieves complexities of $O(\log n)$ time and $O((m+n)\log n)$ work under the arbitrary CRCW PRAM model.

Bader and Cong [5] designed a parallel spanning tree algorithm (denoted as *SPAN_GW*) based on graph traversal for p processors ($p \ll n$). First a small rooted subtree T_s with size $O(p)$ is created. Then each processor picks a unique leaf l from T_s and starts growing a subtree rooted at l by breadth-first traversal. In the process of the traversal, a processor will check a vertex's color, and if it is not colored, color it with a color that is associated with the processor, and set its *parent*. This algorithm is *chaotic* as race conditions among processors are not prevented, and each run may produce a different spanning tree than the last. There are other interesting aspects of the algorithm, for example, work-stealing is employed to dynamically balance the load among processors. We refer interested readers to [5] for details of the algorithm. Here we are interested in the fact that the graph traversal scheme creates a spanning tree after a chaotic execution of the algorithm. With the assumption of sequential consistency, it is shown that the algorithm correctly computes a spanning tree by setting up the parent relation for each vertex in the graph. The core of the proof is lemma 2.

Lemma 2 *On a shared-memory multiprocessor with sequential consistency, no cycles are generated in the spanning tree [5].*

The proof is based on the memory ordering constraints on *SC* that all events are seen on each processor in the same order. A natural question that follows is that whether the same algorithm works for relaxed memory consistency models. The answer depends on, of course, which set of constraints are relaxed.

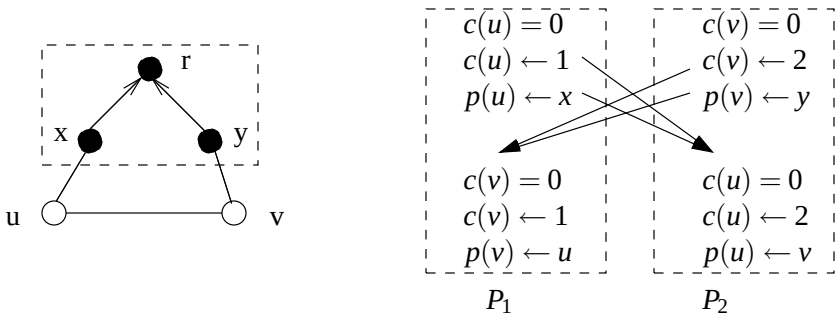

Figure 3: An example where the *SPAN_GW* spanning tree algorithm generates a cycle on *PC*. On the left is a input graph where in the dotted box is the stub spanning tree. Processor P_1 and P_2 are working on vertices *x* and *y*, respectively. On the right are the program sequences executed on P_1 and P_2. Arrows show the propagation of writes.

Here we consider whether the algorithm works for processor consistency that relaxes "write-to-read" ordering. Fig. 3 gives an example where *SPAN_GW* create cycles in the spanning tree if the the underlying memory model provides only processor consistency. In Fig. 3, on the left, immediately after creating a stub spanning tree (including vertices *r*, *x* and *y*), processors P_1 and P_2 start traversing the graph from *x* and *y*, respectively. On the right is a sequence of sequence of events that causes the parent of *u* to be *v* and the parent of *v* to be *u* in the final result. In the sequence, the "=" and "$\leftarrow$" marks are for memory reads and writes, respectively. $c(\cdot)$ is the color of the corresponding vertex, and $p(\cdot)$ is the parent of the vertex. The color for P_1 and P_2 are 1 and 2, respectively. In Fig. 3, on processors P_1 and P_2, memory reads ($c(v) = 0$ and $c(u) = 0$) are executed before preceding writes ($c(u) \leftarrow 1$ and $c(v) \leftarrow 2$) are completed. Scheduling reads before the completion preceding writes results in the formation of cycles in the spanning tree algorithm. It may be possible to design a sequence of operations for *PC* that prevents the formation of cycles, but then it is not clear whether the same sequence works for other relaxed models. Also the proof will be considerably more involved.

Fig. 4 compares the performance of *SPAN_GW* on *SC*, *SPAN_GW* with spinlocks for *RC*, and *SPAN_SV* on *RC*. With spinlocks, in the *SPAN_GW* graph traversal algorithm, a processor waits to enter the critical section that protect the color and parent of a vertex. Races among processors are resolved by mutual exclusions, and there are no multiple processors coloring the same vertex simultaneously. Under both base and optimized hardware implementations, *SPAN_GW* on *SC* runs faster than the other two implementations. Also we see that *SPAN_GW* with spin locks outperform the implementation *SPAN_SV* on *RC*.

Table 2 gives a more detailed performance analysis of the algorithms that run on the base implementation of *SC* and *RC* with 8 processors. Performance metrics such as instructions per cycle (IPC), the percentage of time spent on reads, writes, memory access ordering instructions (Acq and Rel) on *RC*, and barriers, are presented. We include the performance metrics for the implementation of *SPAN_SV* on *SC* to compare with *SPAN_SV_RC*. We see that on *RC SPAN_SV* has higher IPC than on *SC* (0.557 vs. 0.242) as there are fewer consistency constraints with *RC* and the memory operations are better pipelined. Due to the non-structured memory access pattern imposed by the inputs, we see low IPCs and high percentage of time spent on reads/writes for all implementations. *SPAN_GW_SC* has even lower IPC than *SPAN_SV_SC*, yet the overall performance is better as the algorithmic complexity of *SPAN_GW_SC* is lower. It is interesting to compare *SPAN_GW_SC* with the spinlock version – *SPAN_GWLK_RC*. We see that with *SPAN_GWLK_RC* there is a significant amount of time spent on memory access ordering instructions (Acq and Rel) that are executed in the spinlocks. This is one good example that demonstrates the performance advantage of

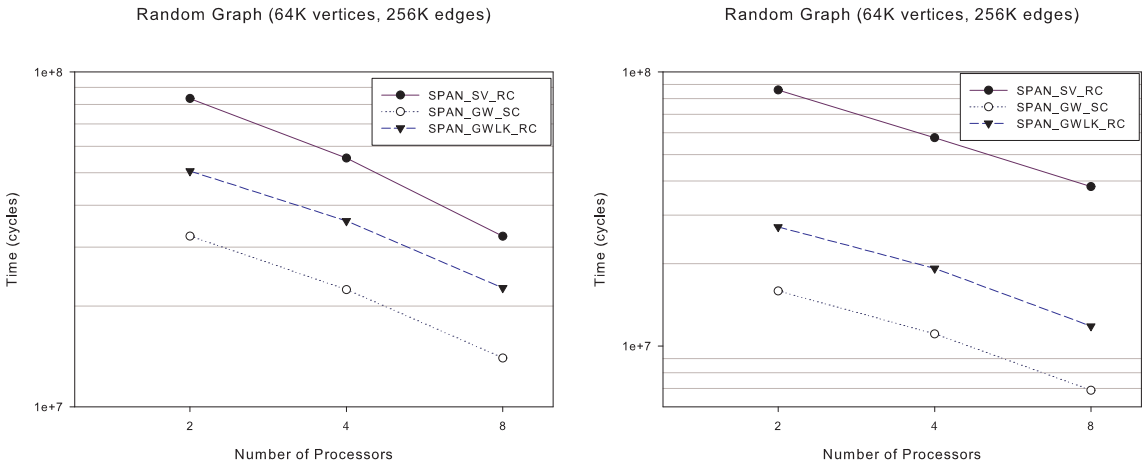

Figure 4: The performance of *SPAN_GW* and *SPAN_SV* on sequential consistency and relaxed models. *SPAN_SV_RC* and *SPAN_GW_SC* are implementations of *SPAN_SV* and *SPAN_GW* on *RC* and *SC*, respectively. *SPAN_GWLK_RC* is the spinlock version of *SPAN_GW* on *RC*.

reasoning with memory constraints on *SC* over mutual exclusion on *RC*. Similar results are observed for optimized implementations of *RC* and *SC*, except that *SPAN_SV* has similar performance metrics on *RC* and *SC*.

Algorithms	IPC	Read	Write	Acq + Rel	Barrier	Other
SPAN_SV_RC	0.55	81.68	0.01	0.00	9.02	9.29
SPAN_SV_SC	0.24	80.34	6.52	0.00	9.34	3.79
SPAN_GW_SC	0.17	70.00	21.19	0.00	5.08	3.72
SPAN_GWLK_RC	0.20	40.97	0.02	53.59	2.46	2.94

Table 2: Performance metrics for *SPAN_GW* and *SPAN_SV* on *SC* and *RC*.

Coordinating multiple processors by reasoning with the ordering constraints on memory operations can also be applied to the design of other algorithms, e.g., parallel MST [6]. An interesting question is whether an arbitrary number of processors can be synchronized under sequential consistency. In fact, it is shown that the ordering constraints with sequential consistency is strong enough so that the mutual exclusion among n processors can be achieved by algorithms such as Lamport's bakery algorithm without special hardware support. More over, Lamport proposed four algorithms that not only provide mutual exclusion for multiple processors, but also allow fault-tolerance properties that deal with the shutdown, abortion, failure, and transient error of processes ([22, 23]). With relaxed models, as there is no longer a single view of the memory access events for all processors, it is hard, if not possible, to infer completion information. As a result, many mutual exclusion algorithms fail to protect critical sections. For example, Both Dijkstra's algorithm and Lamport's bakery algorithm are shown to break under processor consistency (e.g., see [4]).

Mutual exclusion on current shared-memory multiprocessors can also be implemented with atomic hardware operations. With hardware support, spin locks can be constructed with $O(1)$ memory per critical section while both Dijstra's and Lamport's algorithms take $O(p)$ per critical section. For large problems, an extra factor of $O(p)$ in memory use can be very expensive and is in direct proportion to the number of

processors available. In practice most locking primitives employ hardware atomic operations.

5 Conclusions and Future Work

In this paper we studied the impact of memory consistency models on parallel algorithms for shared-memory multiprocessors. We show that the design and implementation of PRAM algorithms are generally oblivious to memory models, and relaxed models that enable aggressive hardware optimizations improve the performance of the benchmark suite consisting of the spanning tree, minimum spanning tree, connected components, biconnected components and list ranking algorithms. Also we presented an example, the spanning tree algorithm based on graph traversal, that reasons with the ordering constraints of *SC* to coordinate processors. The algorithm achieves superior performance over PRAM algorithms on *RC*.

Our results demonstrate the importance of an integrated approach to algorithm and architecture study for parallel computing. We need to design algorithms that map well on to architectures to achieve good performance. At the same time, the design of architecture also benefit from algorithm studies that bring in depth understanding of the problem domain. In terms of memory models, from the perspective of parallel algorithms, we show that relaxed models do not necessarily complicate algorithm designs and algorithms for *SC* can perform better than algorithms for *RC* if the synchronization among processors can be done by reasoning with the memory constraints of *SC*.

The future work includes studying the behavior of the algorithms with more processors, e.g., 16 and 32 processors, exploring the possibility of reasoning with ordering constraints for other problems, and more in-depth comparison of the impact of memory models on different workloads. One related problem is to study whether new, efficient synchronizations can be provided by the architecture (e.g., see [24]), and how it affects the algorithms.

References

[1] S.V. Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Computer science department, University of Wisconsin-Madison, 1993.

[2] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE computer*, 29(12):66–76, 1996.

[3] S.V. Adve, V.S. P, and P. Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. In *proceedings of the IEEE, special issue on distributed shared-memory*, pages 445–455, 1999.

[4] H. Attiya and R. Friedman. Limitations of fast consistency conditions for distributed shared memories. *Information Processing Letters*, 57(5):243–248, 1996.

[5] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, Apr 2004.

[6] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, New Mexico, 2004.

[7] R. Cole and O. Zajicek. The APRAM : incorporating asynchrony into the PRAM model. In *spaa1*, pages 169–178, Jun 1989.

[8] G. Cong and D.A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, Apr 2005.

[9] K. Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Standford University, 1995.

[10] K. Gharachorloo, S.V. Adve, A. Gupta, J.L. Hennessy, and M.D. Hill. Programming for different memory consistency models. *Journal of Parallel and distributed computing*, 15(4):399–407, 1992.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th international conference on architectural support for programming languages and operating system*, pages 245–259, 1991.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 355–364, 1991.

[13] C. Gniady, B. Falsafi, and T.N. Vijaykumar. Is SC + ILP = SC? In *Proceedings of the 26th annual international symposium on computer architecture*, pages 162–171, 1999.

[14] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 16–25, Cape May, NJ, June 1994.

[15] S. Halperin and U. Zwick. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. In *Proc. 7th Ann. Symp. Discrete Algorithms (SODA-96)*, pages 438–447, 1996. Also published in J. Comput. Syst. Sci., 53(3):395–416, 1996.

[16] M.D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8):28–34, 1998.

[17] T.-S. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. In *Proc. 9th Int'l Parallel Processing Symp.*, pages 106–112, Santa Barbara, CA, April 1995.

[18] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[19] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1997.

[20] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[22] L. Lamport. The mutual exclusion poblem: part I - A theory of interprocess communication. *Journal of the ACM*, 33(2):313–325, 1986.

[23] L. Lamport. The mutual exclusion poblem: part II - statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[24] J.F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 18–29, 2002.

[25] V.S. Pai, P. Ranganathan, and S.V. Adve. RSIM: an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessor. In *Proceedings of the 3rd workshop on computer architecture education*, 1997.

[26] P. Ranganathan, V.S. Pai, and S.V. Adve. Using speculative retirement and larger instruction window to narrow the performance gap between memory consistency models. In *Proceedings of the 0th ACM annual symposium on parallel algorithms and architectures (SPAA 97)*, pages 199–210, 1997.

[27] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

[28] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985.

[29] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on computer architecture*, pages 24–36, 1995.