Lock-Free Parallel Algorithms: An Experimental Study

Guojing Cong and David Bader*

Department of Electrical and Computer Engineering,
University of New Mexico, Albuquerque, NM 87131 USA
{cong, dbader}@ece.unm.edu

Abstract. Lock-free shared data structures in the setting of distributed computing have received a fair amount of attention. Major motivations of lock-free data structures include increasing fault tolerance of a (possibly heterogeneous) system and alleviating the problems associated with critical sections such as *priority inversion* and *deadlock*. For parallel computers with tightly-coupled processors and shared memory, these issues are no longer major concerns. While many of the results are applicable especially when the model used is shared memory multiprocessors, no prior studies have considered improving the performance of a parallel implementation by way of lock-free programming. As a matter of fact, often times in practice lock-free data structures in a distributed setting do not perform as well as those that use locks. As the data structures and algorithms for parallel computing are often drastically different from those in distributed computing, it is possible that lock-free programs perform better. In this paper we compare the similarity and difference of lock-free programming in both distributed and parallel computing environments and explore the possibility of adapting lock-free programming to parallel computing to improve performance. Lock-free programming also provides a new way of simulating PRAM and asynchronous PRAM algorithms on current parallel machines.

Keywords: Lock-free Data Structures, Parallel Algorithms, Shared Memory, High-Performance Algorithm Engineering.

1 Introduction

Mutual exclusion locks are widely used for interprocess synchronization due to their simple programming abstractions. However, they have an inherent weakness in a (possibly heterogeneous and faulty) distributed computing environment, that is, the crashing or delay of a process in a critical section can cause deadlock or serious performance degradation of the system [18, 28]. Lock-free data structures (sometimes called concurrent objects) were proposed to allow concurrent accesses of parallel processes (or threads) while avoiding the problems of locks.

* This work was supported in part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

L. Bougé and V.K. Prasanna (Eds.): HiPC 2004, LNCS 3296, pp. 516–527, 2004.

1.1 Previous Results

Lock-free synchronization was first introduced by Lamport in [25] to solve the concurrent *readers and writers problem*. Early work on lock-free data structures focused on theoretical issues of the synchronization protocols, e.g., the power of various atomic primitives and impossibility results [3, 7, 11, 12, 13, 16], by considering the simple *consensus problem* where n processes with independent inputs communicate through a set of shared variables and eventually agree on a common value. Herlihy [20] unified much of the earlier theoretic results by introducing the notion of *consensus number* of an object and defining a hierarchy on the concurrent objects according to their consensus numbers. Consensus number measures the relative power of an object to reach distributed consensus, and is the maximum number of processes for which the object can solve the consensus problem. It is impossible to construct lock-free implementations of many simple and useful data types using any combination of atomic *read, write, test&set, fetch&add* and *memory-to-register swap* because these primitives have consensus numbers either one or two. On the other hand, *compare&swap* and *load-linked, store-conditional* have consensus numbers of infinity, and hence are *universal* meaning that they can be used to solve the consensus problem of any number of processes. Lock-free algorithms and protocols have been proposed for many of the commonly used data structures, e.g., linked lists [37], queues [21, 26, 35], set [27], union-find sets [2], heaps [6], and binary search trees [14, 36]. There are also efforts to improve the performance of lock-free protocols [1, 6].

While lock-free data structures and algorithms are highly resilient to failures, unfortunately, they seem to come at a cost of degraded performance. Herlihy *et al.* studied practical issues and architectural support of implementing lock-free data structures [19, 18], and their experiments with small priority queues show that lock-free implementations do not perform as well as lock-based implementations. LaMarca [24] developed an analytical model based on architectural observations to predict the performance of lock-free synchronization protocols. His analysis and experimental results show that the benefits of guaranteed progress come at the cost of decreased performance. Shavit and Touitou [32] studied lock-free data structures through *software transactional memory*, and their experimental results also show that on a simulated parallel machine lock-free implementations are inferior to standard lock-based implementations.

1.2 Asynchronous Parallel Computation

Cole and Zajichek [9] first introduced lock-free protocols into parallel computing when they proposed asynchronous PRAM (APRAM) as a more realistic parallel model than PRAM because APRAM acknowledges the cost of global synchronization. Their goal was to design APRAM algorithms with fault-resilience that perform better than straightforward simulations of PRAM algorithms on APRAM by inserting barriers. A parallel connected components algorithm without global synchronization was presented as an example. It turned out, however, according to the research of lock-free data structures in distributed computing, that it is impossible to implement many lock-free data structures on APRAM with only atomic register read/write [3, 20]. Attiya *et al.* [4] proved a lower bound of $\log n$ time complexity of any lock-free algorithm on a computational model that is essentially APRAM that achieves *approximate agreement* among n processes in

contrast to constant time of non-lock-free algorithms. This suggests an $\Omega(\log n)$ gap between lock-free and non-lock-free computation models.

Currently lock-free data structures and protocols are still mainly used for fault-tolerance and seem to be inferior in performance to lock-based implementations. In this paper we consider adapting lock-free protocols to parallel computations where multiple processors are tightly-coupled with shared memory. We present novel applications of lock-free protocols where the performance of the lock-free algorithms beat not only lock-based implementations but also the best previous parallel implementations. The rest of the paper is organized as follows: section 2 discusses the potential advantages of using lock-free protocols for parallel computations; section 3 presents two lock-free algorithms as case studies; and section 4 is our conclusion.

2 Application of Lock-Free Protocols to Parallel Computations

Fault-tolerance typically is not a primary issue for parallel computing (as it is for distributed computing) especially when dedicated parallel computers with homogeneous processors are employed. Instead we are primarily concerned with performance when solving large instances. We propose novel applications of lock-free protocols to parallel algorithms that handle large inputs and show that lock-free implementations can have superior performance.

A parallel algorithm often divides into phases and in each phase certain operations are applied to the input with each processor working on portions of the data structure. For irregular problems there usually are overlaps among the portions of data structures partitioned onto different processors. Locks provide a mechanism for ensuring mutually exclusive access to critical sections by multiple working processors. Fine-grained locking on the data structure using system mutex locks can bring large memory overhead. What is worse is that many of the locks are never acquired by more than one processor. Most of the time each processor is working on distinct elements of the data structure due to the large problem size and relatively small number of processors. Yet still extra work of locking and unlocking is performed for each operation applied to the data structure, which results in a large execution overhead.

The access pattern of parallel processors to the shared data structures makes lock-free protocols via atomic machine operations an elegant solution to the problem. When there is work partition overlap among processors, usually it suffices that the overlap is taken care of by one processor. If other processors can detect that the overlap portion is already taken care of, they no longer need to apply the operations and can abort. Atomic operations can be used to implement this "test-and-work" operation. As the contention among processors is low, the overhead of using atomic operations is expected to be small. Note that this is very different from the access patterns to the shared data structures in distributed computing, for example, two producers attempting to put more work into the shared queues. Both producers must complete their operations, and when there is conflict they will retry until success.

In some recent experimental studies on symmetric multiprocessors (SMPs) [14, 36] the design and implementation of lock-free data structures involves mutual-exclusions and are not strictly lock-free in the sense that a crash inside the critical region prevents

application progress. Mutual-exclusion is achieved using inline atomic operations and is transparent to users because it is hidden in the implementation of the data structures. Both Fraser [14] and Valois [36] show that for many search structures (binary tree, skip list, red-black tree) well-implemented algorithms using atomic primitives can match or surpass the performance of lock-based designs in many situations. However, their implementations comprise the guarantee of progress in case of failed processes. "Lock-free" here means free of full-fledged system mutex locks and are actually block-free using spinlocks. Spinlocks do make sense in the homogeneous parallel computing environment with dedicated parallel computers where no process is particularly slower than others and the program is computation intensive. It is a better choice to busy-wait than to block when waiting to enter the critical section. In this paper we also study the application of busy-wait spinlocks to parallel algorithms. We refer interested readers to [30, 35] for examples of block-free data structures.

3 Lock-Free Parallel Algorithms

In this section we present parallel algorithms that are either mutual-exclusion free or block-free to demonstrate the usage of lock-free protocols. Section 3.1 considers the problem of resolving work partition conflicts for irregular problems using a lock-free parallel spanning tree algorithm as an example. Section 3.2 presents an experimental study of a block-free minimum spanning tree (MST) algorithm. As both algorithms take graphs as input, before we present the algorithms, here we describe the the collection of graph generators and the parallel machine we used.

We ran our shared-memory implementations on the Sun Enterprise 4500, a uniform-memory-access shared memory parallel machine with 14 UltraSPARC II processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. The clock speed of each processor is 400 MHz.

Our graph generators include several employed in previous experimental studies of parallel graph algorithms for related problems. For instance, mesh topologies are used in the connected component studies of [15, 17, 22, 23], the random graphs are used by [8, 15, 17, 22], and the geometric graphs are used by [8, 15, 17, 22, 23].

- **Meshes.** Mesh-based graphs are commonly used in physics-based simulations and computer vision. The vertices of the graph are placed on a 2D or 3D mesh, with each vertex connected to its neighbors. **2DC** is a complete 2D mesh; **2D60** is a 2D mesh with the probability of 60% for each edge to be present; and **3D40** is a 3D mesh with the probability of 40% for each edge to be present.
- **Random Graph.** A random graph of n vertices and m edges is created by randomly adding m unique edges to the vertex set. Several software packages generate random graphs this way, including LEDA [29].
- **Geometric Graphs.** Each vertex has a fixed degree k. Geometric graphs are generated by randomly placing n vertices in a unit square and connecting each vertex with its nearest k neighbors. [31] use these in their empirical study of sequential MST algorithms. **AD3** ([23]) is a geometric graph with $k = 3$.

For MST, uniformly random weights are associated with the edges.

3.1 Lock-Free Protocols in Parallel Spanning Tree

We consider the application of lock-free protocols to the Shiloach-Vishkin parallel spanning tree algorithm [33, 34]. This algorithm is representative of several connectivity algorithms that adapt the graft-and-shortcut approach, and is implemented in prior experimental studies. 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.

The algorithm takes an edge list as input and starts with n isolated vertices and m processors. Each processor P_i ($1 \leq i \leq m$) inspects edge $e_i = (v_{i_1}, v_{i_2})$ and tries to graft vertex v_{i_1} to v_{i_2} under the constraint that $v_{i_1} < v_{i_2}$. Grafting creates $k \geq 1$ connected components in the graph, and each of the k components is then shortcutted to to a single supervertex. Grafting and shortcutting are iteratively applied to the reduced graphs $G' = (V', E')$ (where V' is the set of supervertices and E' is the set of edges among supervertices) until only one supervertex is left. For a certain vertex v with multiple adjacent edges, there can be multiple processors attempting to graft v to other smaller vertices. Yet only one grafting is allowed, and we label the corresponding edge that causes the grafting as a spanning tree edge. This is a partition conflict problem.

Two-phase election can be used to resolve the conflicts. The strategy is to run a race among processors, where each processor that attempts to work on a vertex v writes its id into a tag associated with v. After all the processors are done, each processor checks the tag to see whether it is the winning processor. If so, the processor continues with its operation, otherwise it aborts. Two-phase election works on platforms that provide write atomicity. A global barrier synchronization among processors is used instead of a possibly large number of fine-grained locks. The disadvantage is that two runs are involved.

A natural solution to the work partition problem is to use lock-free atomic instructions. When a processor attempts to graft vertex v, it invokes the atomic *compare&swap* operation to check whether v has been worked on. If not, the atomic nature of the operation also ensures that other processors will not work on v again. The detailed description of the algorithm and an inline assembly function for *compare&swap* can be found in [10].

We compare the performance of the lock-free Shiloach-Vishkin spanning tree implementation with four other implementations that differ only in how the conflicts are resolved. In Table 1 we describe the four implementations.

Table 1. Five implementations of Shiloach-Vishkin's parallel spanning tree algorithm

Implementation	Description
span-2phase	conflicts are resolved by two-phase election
span-lock	conflicts are resolved using system mutex locks
span-lockfree	no mutual exclusion, races are prevented by atomic updates
span-spinlock	mutual exclusion by spinlocks using atomic operations
span-race	no mutual exclusion, no attempt to prevent races

Among the four implementations, **span-race** is not a correct implementation and does not guarantee correct results. It is included as a baseline to show how much overhead is involved with using lock-free protocols and spinlocks.

Experimental Results. In Fig. 1 we see **span-lock** does not scale with the number of the processors, and is consistently the approach with the worst performance. **span-2phase**, **span-lockfree**, and **span-spinlock** scale well with the number of processors, and the execution time of **span-lockfree** and **span-spinlock** is roughly half of that of **span-2phase**. It is interesting to note that **span-lockfree**, **span-spinlock** and **span-race** are almost as fast as each other for various inputs, which suggests similar overhead for spinlocks and lock-free protocols, and the overhead is negligible.

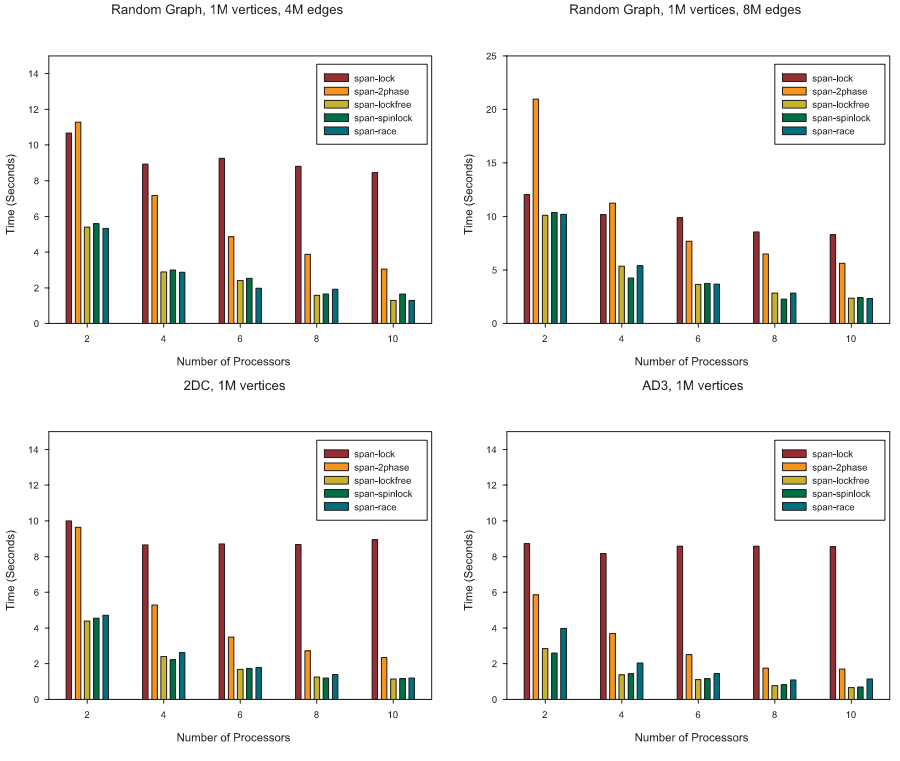

Fig. 1. The performance of the spanning tree implementations. The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively

3.2 Block-Free Parallel Algorithms

For parallel programs that handle large inputs on current SMPs, spinlocks are a better choice than the blocking system mutex locks. Spinlocks are simpler, take less memory and do not involve the kernel. Due to the large inputs and relatively smaller number

of processors available, most of the time each processor is working on distinct data elements, and the contention over a certain lock or data element is very low. Even in case of contention, as the expected time that a processor spends in the critical section is short, it is much cheaper to busy wait for a few cycles than to block. In the previous experimental study with the spanning tree algorithm, we have already seen that spinlock is a good candidate for mutual exclusion. As an example we next present a parallel minimum spanning tree (MST) algorithm using spinlocks for synchronization that achieves a more drastic performance improvement.

Parallel Borůvka's Algorithm and Previous Experimental Studies. Given an undirected connected graph G with n vertices and m edges, the minimum spanning tree (MST) problem finds a spanning tree with the minimum sum of edge weights. In our previous work [5], we studied the performance of different variations of parallel Borůvka's algorithm. Borůvka's algorithm is comprised of Borůvka iterations that are used in many parallel MST algorithms. A Borůvka iteration is characterized by three steps: *find-min*, *connected-components* and *compact-graph*. In *find-min*, for each vertex v the incident edge with the smallest weight is labeled to be in the MST; *connect-components* identifies connected components of the induced graph with the labeled MST edges; *compact-graph* compacts each connected component into a single supervertex, removes self-loops and multiple edges, and re-labels the vertices for consistency.

Here we summarize each of the Borůvka algorithms. The major difference among them is the input data structure and the implementation of *compact-graph*. **Bor-ALM** takes an adjacency list as input and compacts the graph using two parallel sample sorts plus sequential merge sort; **Bor-FAL** takes our *flexible adjacency list* as input and runs parallel sample sort on the vertices to compact the graph. For most inputs, **Bor-FAL** is the fastest implementation. In the *compact-graph* step, **Bor-FAL** merges each connected components into a single supervertex that gets the adjacency list of all the vertices in the component. **Bor-FAL** does not attempt to remove self-loops and multiple edges, and avoids runs of extensive sortings. Self-loops and multiple edges are filtered out in the *find-min* step instead. **Bor-FAL** greatly reduces the number of shared memory writes at the relatively small cost of an increased number of reads, and proves to be efficient as predicted on current SMPs.

A New Implementation. Now we present an implementation with spinlocks (denoted as **Bor-spinlock**) that further reduces the number of memory writes. In fact the input edge list is not modified at all in **Bor-spinlock**, and the *compact-graph* step is completely eliminated. The main idea is that instead of compacting connected components, for each vertex there is now an associated label *supervertex* showing to which supervertex it belongs. In each iteration all the vertices are partitioned among the processors. For each vertex v of its assigned partition, processor p finds the adjacent edge e with the smallest weight. If we compact connected components, e would belong to the supervertex v' of v in the new graph. Essentially processor p finds the adjacent edge with smallest weight for v'. As we do not compact graphs, the adjacent edges for v' are scattered among the adjacent edges of all vertices that share the same supervertex v', and different processors may work on these edges simultaneously. Now the problem is that these processors need to synchronize properly in order to find the edge with the mini-

mum weight. Again this is an example of the irregular work-partition problem. Fig. 2 illustrates the specific problem for the MST case.

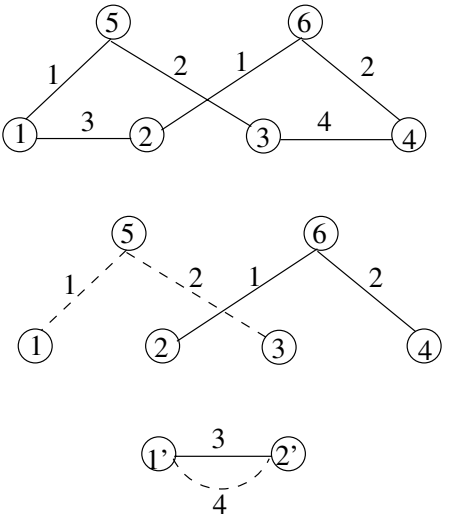

Fig. 2. Example of the race condition between two processors when Borůvka's algorithm is used to solve the MST problem

On the top in Fig. 2 is an input graph with six vertices. Suppose we have two processors P_1 and P_2. Vertices 1, 2, and 3, are partitioned on to processor P_1 and vertices 4, 5, and 6 are partitioned on to processor P_2. It takes two iterations for Borůvka's algorithm to find the MST. In the first iteration, the *find-min* step of **Bor-spinlock** labels $< 1,5 >$, $< 5,3 >$, $< 2,6 >$ and $< 6,4 >$ to be in the MST. *connected-components* finds vertices 1, 3, and 5, in one component, and vertices 2, 4, and 6, in another component. The MST edges and components are shown in the middle of Fig. 2. Vertices connected by dashed lines are in one component, and vertices connected by solid lines are in the other component. At this time, vertices 1, 3, and 5, belong to supervertex $1'$, and vertices 2, 4, and 6, belong to supervertex $2'$. In the second iteration, processor P_1 again inspects vertices 1, 2, and 3, and processor P_2 inspects vertices 4, 5, and 6. Previous MST edges $< 1,5 >$, $< 5,3 >$, $< 2,6 >$ and $< 6,4 >$ are found to be edges inside supervertices and are ignored. On the bottom in Fig. 2 are the two supervertices with two edges between them. Edges $< 1,2 >$ and $< 3,4 >$ are found by P_1 to be the edges between supervertices $1'$ and $2'$, edge $< 3,4 >$ is found by P_2 to be the edge between the two supervertices. For supervertex $2'$, P_1 tries to label $< 1,2 >$ as the MST edge while P_2 tries to label $< 3,4 >$. This is a race condition between the two processors, and locks are used in **Bor-spinlock** to ensure correctness. The formal description of the algorithm is given in [10].

We compare the performance of **Bor-spinlock** with the best previous parallel implementations. The results are shown in Fig. 3. **Bor-FAL** is the fastest implementation for sparse random graphs, **Bor-ALM** is the fastest implementation for meshes. From

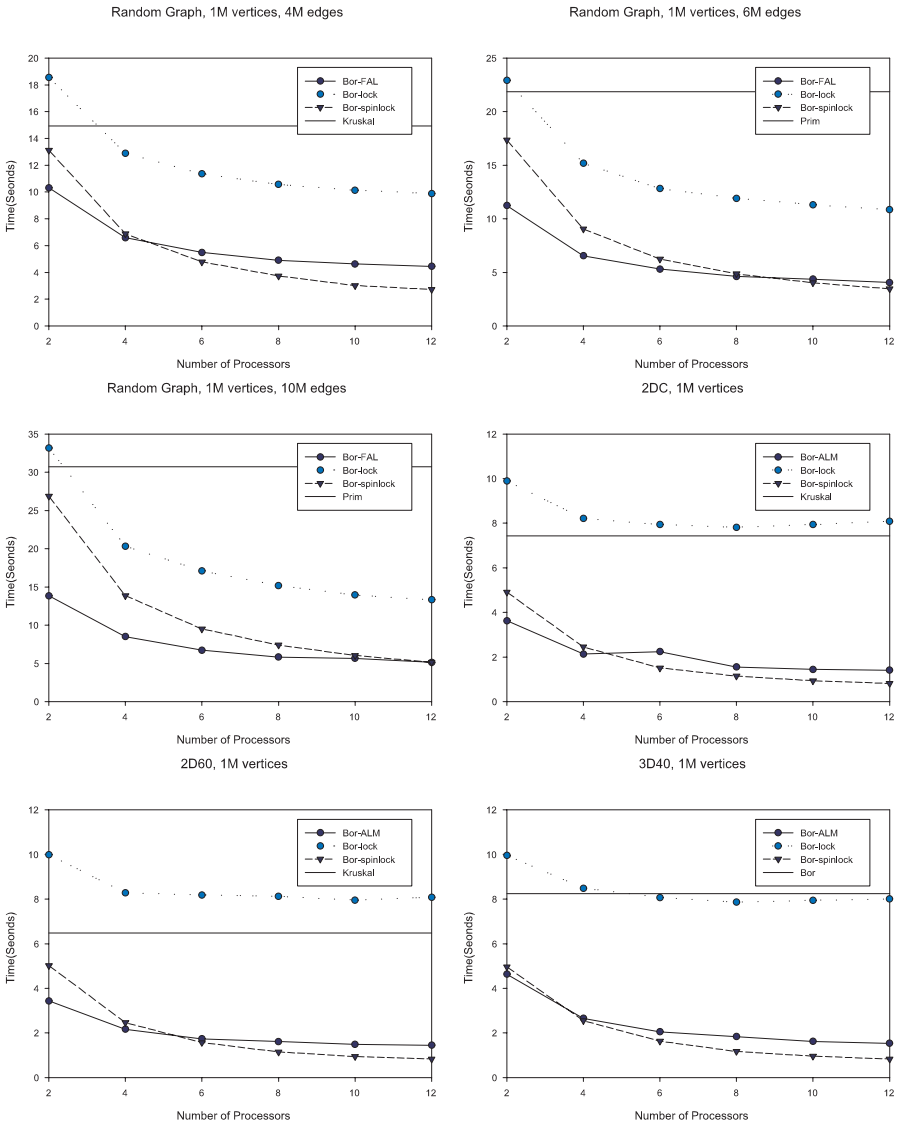

Fig. 3. Comparison of the performance of **Bor-spinlock** against the previous implementations. The horizontal line in each graph shows the execution time of the best sequential implementation

our results we see that with 12 processors **Bor-spinlock** beats both **Bor-FAL** and **Bor-ALM**, and performance of **Bor-spinlock** scales well with the number of processors. In Fig. 3, performance of **Bor-lock** is also plotted. **Bor-lock** is the same as **Bor-spinlock** except that system mutex locks are used. **Bor-lock** does not scale with the number of processors. The performance of the best sequential algorithms among the three candidates, Kruskal, Prim, and Borůvka, is plotted as a horizontal line for each input graph.

For all the input graphs shown in Fig. 3, **Bor-spinlock** tends to perform better than the previous best implementations when more processors are used. Note that a maximum speedup of 9.9 for 2D60 with 1M vertices is achieved with **Bor-spinlock** at 12 processors. Fig. 3 demonstrates the potential advantage of spinlock-based implementations for large and irregular problems. Aside from good performance, **Bor-spinlock** is also the simplest approach as it does not involve sorting required by the other approaches.

4 Conclusions

In this paper we present novel applications of lock-free and block-free protocols to parallel algorithms and show that these protocols can greatly improve the performance of parallel algorithms for large, irregular problems. As there is currently no direct support for invoking atomic instructions from most programming languages, our results suggest it necessary that there be orchestrated support for high performance algorithms from the hardware architecture, operating system, and programming languages. Two graph algorithms are discussed in this paper. In our future work, we will consider applying lock-free and block-free protocols to other types of algorithms, for example, parallel branch-and-bound.

References

1. J. Allemany and E.W. Felton. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proc. 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134, Aug 1992.
2. R.J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 370 – 380, May 1991.
3. J. Aspnes and M. P. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd Ann. Symp. Parallel Algorithms and Architectures (SPAA-90)*, pages 340–349, Jul 1990.
4. H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, 1994.
5. D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, page to appear, Santa Fe, New Mexico, April 2004.
6. G. Barnes. Wait free algorithms for heaps. Technical Report TR-94-12-07, University of Washington, 1994.
7. B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, Vancouver, British Columbia, Canada, 1987.
8. S. Chung and A. Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. In *Proc. 10th Int'l Parallel Processing Symp. (IPPS'96)*, pages 302–315, April 1996.
9. R. Cole and O. Zajicek. The APRAM : incorporating asynchrony into the PRAM model. In *Proc. 1st Ann. Symp. Parallel Algorithms and Architectures (SPAA-89)*, pages 169–178, Jun 1989.
10. G. Cong and D.A. Bader. Lock-free parallel algorithms: an experimental study. Technical report, Electrical and Computer Engineering Dept., University of Mexico, 2004.

11. C. Dwork, D. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.
12. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
13. M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *J. ACM*, 32(2):374–382, 1985.
14. K.A. Fraser. *Practical lock-freedom.* PhD thesis, King's College, University of Cambridge, Sep 2003.
15. S. Goddard, S. Kumar, and J.F. Prins. Connected components algorithms for mesh-connected parallel computers. 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 43–58. American Mathematical Society, 1997.
16. A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer — designing a MIMD, shared-memory parallel machine. *IEEE Trans. Computers*, C-32(2):175–189, 1984.
17. 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.
18. M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Int'l Symposium in Computer Architecture*, pages 289–300, May 1993.
19. M.P. Herlihy. A methodology for implementing highly concurrent data objects. In *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Mar 1990.
20. M.P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
21. M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *Proc. 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 13 – 26, Jan 1987.
22. T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. 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 23–41. American Mathematical Society, 1997.
23. 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.
24. A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proc. 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 130 – 140, Aug 1994.
25. L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.
26. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.
27. V. Lanin and D. Shaha. Concurrent set manipulation without locking. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 211 – 220, Mar 1988.
28. H. Massalin and C. Pu. Threads and input/output in the synthesis kernel. In *Proc. 12th ACM Symposium on Operating Systems Principles*, pages 191–201, Dec 1989.

29. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
30. M.M. Michael and M.L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.
31. B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics* ***15***, pages 99–117. American Mathematical Society, 1994.
32. N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug 1995.
33. Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.
34. R.E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.
35. P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proc. 13th Ann. Symp. Parallel Algorithms and Architectures (SPAA-01)*, pages 134–143, Sep 2001.
36. J. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, May 1995.
37. J.D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214 – 222, Aug 1995.