Designing Irregular Parallel Algorithms With Mutual Exclusion and Lock-free Protocols

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

David A. Bader*
College of Computing
Georgia Institute of Technology

February 25, 2006

Abstract

Irregular parallel algorithms pose a significant challenge for achieving high performance because of the difficulty predicting memory access patterns or execution paths. Within an irregular application, fine-grained synchronization is one technique for managing the coordination of work; but in practice the actual performance for irregular problems depends on the input, the access pattern to shared data structures, the relative speed of processors, and the hardware support of synchronization primitives. In this paper, we focus on lock-free and mutual exclusion protocols for handling fine-grained synchronization. Mutual exclusion and lock-free protocols have received a fair amount of attention in coordinating accesses to shared data structures from concurrent processes. Mutual exclusion offers a simple programming abstraction, while lock-free data structures provide better fault tolerance and eliminate problems associated with critical sections such as *priority inversion* and *deadlock*. These synchronization protocols, however, are seldom used in parallel algorithm designs, especially for algorithms under the SPMD paradigm, as their implementations are highly hardware dependent and their costs are hard to characterize. Using graph-theoretic algorithms for illustrative purposes, we show experimental results on two shared-memory multiprocessors, the IBM pSeries 570 and the Sun Enterprise 4500, that irregular parallel algorithms with efficient fine-grained synchronization may yield good performance.

*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, DBI-0420513, ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

1 Introduction

Irregular problems are challenging to parallelize and to achieve high performance because typically their memory access patterns or execution paths are not predictable *a priori*, and straightforward data decompositions or load balancing techniques, such as those used for regular problems, often are not efficient for these applications. Fine-grained synchronization, a technique for managing the coordination of work within an irregular application, can be implemented through lock-free protocols, system mutex locks, and spinlocks. However, fine-grained locks and lock-free protocols are seldomly employed in implementations of parallel algorithms.

System mutex locks, widely used for interprocess synchronization due to their simple programming abstractions, provide a common interface for synchronization, and the performance depends on the implementation and the application scenario. User-defined spinlocks are customizable; however, the disadvantages are that the user is exposed to low-level hardware details and portability can be an issue. For large-scale application of locks in a high-performance computing environment, spinlocks have the advantage of economic memory usage and simple instruction sequences. Each spinlock can be implemented using one memory word, while a system mutex lock can take multiple words for its auxiliary data structures, which exacerbates the problem with accessing memory.

Mutual exclusion locks 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 [30, 46]. 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. In theory we can coordinate any number of processors through lock-free protocols. In practice, however, lock-free data structures are primarily used for fault-tolerance.

In this paper, we illustrate the performance of fine-grained locks and lock-free protocols using irregular applications such as those from graph theory, using large, sparse instances on shared-memory multiprocessors. Graph abstractions are used in many computationally challenging science and engineering problems. For instance, the minimum spanning tree

(MST) problem finds a spanning tree of a connected graph G with the minimum sum of edge weights. MST is one of the most studied combinatorial problems with practical applications in VLSI layout, wireless communication, and distributed networks [48, 59, 67], recent problems in biology and medicine such as cancer detection [12, 37, 38, 47], medical imaging [2], and proteomics [52, 23], and national security and bioterrorism such as detecting the spread of toxins through populations in the case of biological/chemical warfare [13], and is often a key step in other graph problems [50, 45, 58, 64]. Graph abstractions are also used in data mining, determining gene function, clustering in semantic webs, and security applications. For example, studies (e.g. [16, 39]) have shown that certain activities are often suspicious not because of the characteristics of a single actor, but because of the interactions among a group of actors. Interactions are modeled through a graph abstraction where the entities are represented by vertices, and their interactions are the directed edges in the graph. This graph may contain billions of vertices with degrees ranging from small constants to thousands. Due to our interest in large graphs, we explore the performance of graph applications that use a tremendous number (e.g., millions to billions) of fine-grained synchronizations.

Most theoretic parallel algorithmic models are either synchronous (e.g., PRAM [36]) or for network-based systems (e.g., LogP [21] and BSP [61]) with no explicit support for fine-grained synchronization. In these models, coarse synchronization is performed through a variety of mechanisms such as lock-step operation (as in PRAM), algorithm supersteps (as in BSP) and collective synchronization primitives such as barriers (as in LogP), rather than fine-grained coordination of accesses to shared data structures. In practice, the performance of parallel algorithms that use locks and lock-free protocols are highly dependent on the parallel computer architecture and the contention among processors to shared regions.

In this paper we investigate the performance of fine-grained synchronization on irregular parallel algorithms using shared-memory multiprocessors. These high-performance parallel systems typically have global access to large, shared memories and avoid the overhead of explicit message passing. Fast parallel algorithms for irregular problems have been developed for such systems. For instance, we have designed fast parallel graph algorithms and demonstrated speedups compared with the best sequential implementation for problems such as ear

decomposition [9], tree contraction and expression evaluation [10], spanning tree [6, 8], rooted spanning tree [20], and minimum spanning forest [7]. Many of these algorithms achieve good speedups due to algorithmic techniques for efficient design and better cache performance. For some of the instances, for example, arbitrary, sparse graphs, while we may be able to improve the cache performance to a certain degree, there are no known general techniques for cache performance optimization because the memory access pattern is largely determined by the structure of the graph. Our prior studies have excluded certain design choices that involve fine-grained synchronizations. This paper investigates these design choices with lock-free protocols and mutual exclusion. Our main results include novel applications of fine-grained synchronization where the performance beats the best previously-known parallel implementations.

The rest of the paper is organized as follows: Section 2 presents lock-free parallel algorithms with an example of lock-free spanning tree algorithm; Section 3 presents parallel algorithms with fine-grained locks; Section 4 compares the performance of algorithms with fine-grained synchronizations with prior implementations; and Section 5 gives our conclusions and future work.

2 Lock-free Parallel Algorithms

Lamport [42] first introduced lock-free synchronization to solve the concurrent *readers and writers problem* and improve fault-tolerance. Before we present its application to the design of parallel algorithms, we first give a brief review of lock-free protocols and some theoretic results in Section 2.1. Section 2.2 summarizes prior results on lock-free parallel algorithms.

2.1 Lock-free Shared Data Structures

Early work on lock-free data structures focused on theoretical issues of the synchronization protocols, for example, the power of various atomic primitives and impossibility results [4, 14, 22, 24, 25, 28], 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 [31] 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 are proposed for many commonly-used data structures, for example, linked lists [63], queues [33, 43, 60], set [44], union-find sets [3], heaps [11], and binary search trees [26, 62]; and also for the performance improvement of lock-free protocols [1, 11]. 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 [32, 30], and their experiments with small priority queues show that lock-free implementations do not perform as well as lock-based implementations. With 16 processors on an Encore Multimax, the lock-free implementation, for a benchmark that enqueues and dequeues $1M$ elements, with exponential back-off to reduce contention is about 30% slower than the corresponding lock-based implementation. (Note that throughout this paper, we use M to represent 2^{20}.) LaMarca [41] developed an analytic 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 [55] 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.

2.2 Asynchronous Parallel Computation

Cole and Zajicek [19] 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 [4, 31]. Attiya *et al.* [5] 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. Vishkin *et al.* introduced the "independence of order semantics (IOS)" that provides lock-free programming on explicit multi-threading (XMT) [65].

2.3 Lock-free Protocols for Resolving Races among Processors

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 may result in a large execution overhead depending on the implementation of locks.

We consider the following problem. For a given input array A of size n, a certain operation

op from a set of operations is to be applied to the elements in A according to the conditions specified in the condition array C of size m with $m \geq n$. $C(j)$ $(1 \leq j \leq m)$ specifies an element $A(i)$, a condition $cond$ which may be a Boolean expression involving elements of A, and an operation op; if $cond$ is evaluated as true, operation op is applied to $A(i)$. In case multiple conditions in C for a certain element $A(i)$ are satisfied, there is a potential race condition for all processors as applying the operation involves the evaluation of the condition (which, in general, is not atomic). Depending on different algorithms, either one operation or a certain subset of the operations are applied. Here we consider the case when only one arbitrary operation is applied.

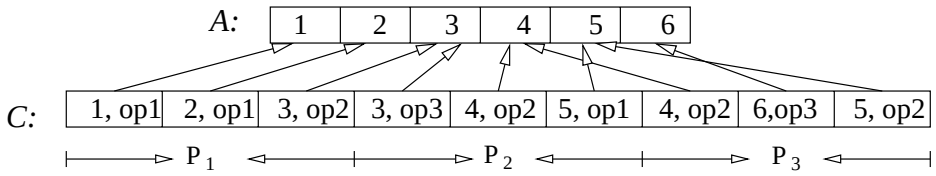

Figure 1: Conflicts when partitioning work among the processors. Here for simplicity we assume every condition in C is true and they are not shown.

In Fig. 1, array C is partitioned onto three processors P_1, P_2, and P_3. Processor P_1 and P_2 will both work on $A(3)$, and P_2 and P_3 will both work on $A(4)$. To resolve the conflicts among processors, we can either sort array C, which is expensive, to move the conditions for $A(i)$ into consecutive locations and guarantee that only one processor works on $A(i)$ or use fine-grained synchronization to coordinate multiple processors.

Here we show that lock-free protocols via atomic machine operations are an elegant solution to the problem. When there is work partition overlap among processors, 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, we expect the overhead of using atomic operations 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.

To illustrate this point in a concrete manner, we consider the application of lock-free protocols to the Shiloach-Vishkin parallel spanning tree algorithm [56, 57]. This algorithm is representative of several connectivity algorithms that adapt the graft-and-shortcut approach, and is implemented in prior experimental studies (e.g., see [29, 40, 34]). 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 $i_1 < i_2$. Grafting creates $k \geq 1$ connected components in the graph, and each of the k components is then shortcut 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-labeled 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 is one method that 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 processor 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. 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.

Another more 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 on whether v has been inspected. 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 is shown in Alg. 1, and inline assembly functions for *compare&swap* can be found in Algs. 2 and 3 in Section 2.4.

Data : (1) EdgeList$[1 \ldots 2m]$: edge list representation for graph $G = (V, E)$, $|V| = n$, $|E| = m$; each element of EdgeList has two field, v_1 and v_2 for the two endpoints of an edge
(2) integer array D$[1 \ldots n]$ with D$[i] = i$
(3) integer array Flag$[1 \ldots n]$ with Flag$[i] = 0$

Result : a sequence of edges that are in the spanning tree
begin
  $n' = n$
  while $n' \neq 1$ **do**
    for $k \leftarrow 1$ *to* n' **in parallel do**
      $i =$ EdgeList$[k].v_1$
      $j =$ EdgeList$[k].v_2$
      if D$[j] <$ D$[i]$ *and* D$[i] =$ D[D$[i]]$ *and compare&swap*(&Flag[D$[i]], 0, PID$) = 0 **then**
        label edge EdgeList$[k]$ to be in the spanning tree
        D[D$[i]] =$ D$[j]$
    for $i \leftarrow 1$ *to* n' **in parallel do**
      while D$[i] \neq$ D[D$[i]] **do**
        D$[i] =$ D[D$[i]]$
    $n' =$ the number of super-vertices
end

Algorithm 1: Parallel Lock-Free Spanning Tree Algorithm (**span-lockfree**)

2.4 Implementation of Compare&Swap

```
.inline compare&swap
 cas [%o0], %o1, %o2
 mov %o2, %o0
.end
```

Algorithm 2: The *compare&swap* function implementation on Sun Sparc.

As atomic instructions are generally not directly available to high level programming languages, we show in Alg. 2 the design of an atomic *compare&swap* instruction in an inline C function for Sun Sparc. In the example, [*o0*] stands for the address held in register *o0*. On Sun Sparc, the `cas` instruction compares the word at memory address [*o0*] and the word in register *o1*. If they are the same, then the word in register *o2* and word are swapped; otherwise no swapping is done but *o2* still receives the value stored in [*o0*].

```
#pragma mc_func compare&swap {           \
 "7cc01828"  /* cas_loop: lwarx 6,0,3 */ \
 "7c043000"  /* cmpw 4,6 */              \
 "4082000c"  /* bc 4,2,cas_exit */       \
 "7ca0192d"  /* stwcx. 5,0,3 */          \
 "4082fff0"  /* bc 4,2,cas_loop */       \
 "7cc33378"  /* cas_exit: or 3,6,6 */    \
}
#pragma reg_killed_by CASW  gr0,gr3,gr4,gr5,gr6,cr0
```

Algorithm 3: The *compare&swap* function implementation using *load-linked, store-conditional* on PowerPC

For the IBM PowerPC architecture, Alg. 3 demonstrates the *compare&swap* implemented through *load-linked, store-conditional* instructions. Inline assembly is not directly supported with IBM's native C compiler. Instead, the assembly code is first translated into machine code and then linked. In the example, the comments show the corresponding assembly code for the machine code.

In this example, the pair of instructions `lwarx` and `stwcx.` are used to implement a read-

modify-write operation to memory. Basically `lwarx` is a special load, and `stwcx.` a special store. If the store from a processor is performed, then no other processor or mechanism has modified the target memory location between the time the `lwarx` instruction is executed and the time the `stwcx.` instruction completes.

3 Parallel Algorithms with Fine-grained Mutual Exclusion Locks

Mutual exclusion provides an intuitive way for coordinating synchronization in a parallel program. For example, in the spanning algorithm in Section 2.3, we can also employ mutual exclusion locks to resolve races among processors. Before a processor grafts a subtree that is protected by critical sections, it first gains access to the data structure by acquiring locks, which guarantees that a subtree is only grafted once. In Section 3.1 we discuss the design and implementation of spinlocks for mutual exclusion.

To illustrate the use of mutex locks, in this section we present a new implementation of the minimum spanning tree (MST) problem based on parallel Borůvka's algorithm that outperforms all previous implementations. We next introduce parallel Borůvka's algorithm and previous experimental results.

3.1 Implementation of Spinlocks

Implementations of spinlocks on Sun Sparc and IBM PowerPC are shown in Algs. 4 and 5, respectively. Note that `cas` and *load-linked,store-conditional* are used. In addition, there are also memory access synchronizing instructions. For example `membar` on Sparc and `sync` on PowerPC, are employed to guarantee consistency in relaxed memory models which are related to the implementation of synchronization primitives, but are largely outside the scope of this paper. We refer interested readers to [66] and [35] for documentation on atomic operations and memory synchronization operations.

```
.inline spin_lock
1:
mov 0,%o2
cas [%o0],%o2,%o1
tst %o1
be 3f
nop
2:
ld [%o0], %o2
tst %o2
bne 2b
nop
ba,a 1b
3:
membar #LoadLoad | #LoadStore
.end

.inline spin_unlock
membar #StoreStore
membar #LoadStore !RMO only
mov 0, %o1
st %o1,[%o0]
.end
```

Algorithm 4: The implementation of a spinlock on Sun Sparc.

```
#pragma mc_func spin_lock{                    \
 "7cc01828" /* TRY: lwarx 6, 0, 3 */         \
 "2c060000" /* cmpwi 6,0 */                  \
 "4082fff8" /* bc 4,2,TRY */                 \
 "4c00012c" /* isync */                      \
 "7c80192d" /* stwcx. 4, 0, 3 */             \
 "4082ffec" /* bc 4,2, TRY */                \
 "4c00012c" /* isync ## instruction sync */ \
}
#pragma reg_killed_by spin_lock gr0, gr3, gr4, gr6

#pragma mc_func spin_unlock{                  \
 "7c0004ac" /* sync */                       \
 "38800000" /* addi 4, 0, 0 */               \
 "90830000" /* stw 4,0(3) */                 \
}
#pragma reg_killed_by spin_unlock gr0,gr3, gr4
```

Algorithm 5: The implementation of a spinlock on IBM PowerPC.

3.2 Parallel Borůvka's Algorithm

Given an undirected connected graph G with n vertices and m edges, the minimum spanning tree problem finds a spanning tree with the minimum sum of edge weights. In our previous work [7], we studied the performance of different variations of parallel Borůvka's algorithm. Borůvka's algorithm is comprised of Borůvka iterations that are used in several parallel MST algorithms (e.g., see [53, 54, 18, 17]). 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*. The *compact-graph* is the most expensive of the three steps. **Bor-ALM** takes an adjacency list as input and

compacts the graph using parallel sample sort 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 combines 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. Instead, self-loops and multiple edges are filtered out in the *find-min* step. **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.

3.3 A New Implementation with Fine-grained Locks

Now we present an implementation with fine-grained locks that further reduces the number of memory writes. In fact the input edge list is not modified at all in the new implementation, 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 as evenly as possible 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 minimum weight. Again this is an example of the irregular work-partition problem. Fig. 2 illustrates the specific problem for the MST case.

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 labels $\langle 1, 5\rangle$, $\langle 5, 3\rangle$, $\langle 2, 6\rangle$, and $\langle 6, 4\rangle$, to be

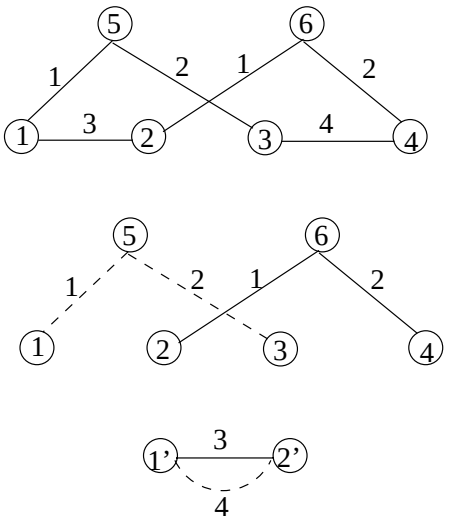

Figure 2: Example of the race condition between two processors when Borůvka's algorithm is used to solve the MST problem.

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 $\langle 1, 5\rangle$, $\langle 5, 3\rangle$, $\langle 2, 6\rangle$ and $\langle 6, 4\rangle$ are found to be edges inside supervertices and are ignored. On the bottom of Fig. 2 are the two supervertices with two edges between them. Edges $\langle 1, 2\rangle$ and $\langle 3, 4\rangle$ are found by P_1 to be the edges between supervertices $1'$ and $2'$, edge $\langle 3, 4\rangle$ is found by P_2 to be the edge between the two supervertices. For supervertex $2'$, P_1 tries to label $\langle 1, 2\rangle$ as the MST edge while P_2 tries to label $\langle 3, 4\rangle$. This is a race condition between the two processors, and locks are used in to ensure correctness. The formal description of the algorithm is given in Alg. 6. Note that Alg. 6 describes the parallel MST algorithm with generic locks. The locks in the algorithm can be either replaced by system mutex locks or spinlocks.

Depending on which types of locks are used, we have two implementations, **Bor-spinlock** with spinlocks and **Bor-lock** with system mutex locks. We compare their performance with the best previous parallel implementations in Section 4.

```
Data   : (1) graph G = (V, E) with adjacency list representation, |V| = n
         (2) array D[1...n] with D[i] = i
         (3) array Min[1...n] with Min[i] = MAXINT
         (4) array Graft[1...n] with Graft[i] = 0

Result : array E_MST of size n − 1 with each element being a MST tree edge
begin
    while not all D[i] have the same value do
        for i ← 1 to n in parallel do
            for each neighbor j of vertex i do
                if D[i] ≠ D[j] then
                    lock(D[i])
                    if Min[D[i]] < w(i, j) then
                        Min[D[i]] ← w(i, j)
                        Graft[D[i]] ← D[j]
                        Record/update edge e = ⟨i, j⟩ with the minimum weight
                    unlock(D[i])
        for i ← 1 to n in parallel do
            if Graft[i] ≠ 0 then
                D[i] ← Graft[i]
                Graft[i] ← 0
                Min[i] ← MAXINT
                Retrieve the edge e that caused the grafting
                Append e to the array E_MST
        for i ← 1 to n in parallel do
            while D[i] ≠ D[D[i]] do
                D[i] ← D[D[i]]
end
```

Algorithm 6: Parallel Borůvka Minimum Spanning Tree Algorithm

4 Experimental Results

We ran our shared-memory implementations on two platforms, the Sun Enterprise 4500 (E4500) and IBM pSeries 570 (p570). They are both uniform-memory-access shared memory parallel machines. The Sun Enterprise 4500 system has 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. The IBM p570 has 16 IBM Power5 processors and 32 GB of memory, with 32 Kbytes L1 data cache, 1.92 Mbytes L2 cache. There is a L3 cache with 36 Mbytes per two processors. The processor clock speed is 1.9 GHz.

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 [27, 29, 34, 40], the random graphs are used by [15, 27, 29, 34], and the geometric graphs are used by [15, 27, 29, 34, 40].

- *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 [49].
- **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. Moret and Shapiro [51] use these in their empirical study of sequential MST algorithms. **AD3** (used by Krishnamurthy *et al.* in [40]) is a geometric graph with $k = 3$.

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

Before discussing experimental results for spanning tree and MST algorithms in Sections 4.2 and 4.3, we show that for large data inputs, algorithms with fine-grained synchronizations do not incur serious contention among processors.

4.1 Contention Among Processors

With fine-grained parallelism, contention may occur for access to critical sections or to memory locations in shared data structures. The amount of contention is dependent on the problem size, number of processors, memory access patterns, and execution times for regions of the code. In this section, we investigate contention for our fine-grained synchronization methods and quantify the amount of contention in our graph theoretic example codes.

To measure contention, we record the number of times a spinlock spins before it gains access to the shared data structure. For lock-free protocols it is difficult to measure the actual contention. For example, if *compare&swap* is used to partition the workload, it is impossible to tell whether the failure is due to contention from another contending processor or due to the fact that the location has already been claimed before. However, inspecting how spinlocks behave can give a good indication of the contention for lock-free implementations as in both cases processors contend for the same shared data structures.

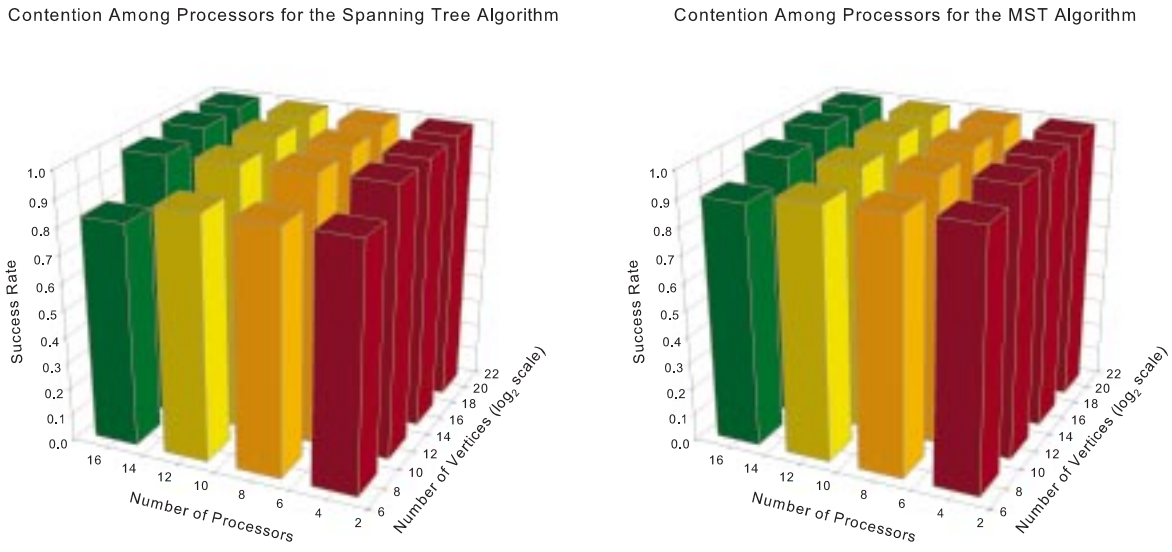

Figure 3: Contention among processors for **span-spinlock** and **Bor-spinlock**. The input graphs are random graphs with n vertices and $4n$ edges.

Fig. 3 shows the contention among processors for the spanning tree and MST algorithms with different number of processors and sizes of inputs. The level of contention is represented by *success rate*, which is calculated as the total number of locks acquired divided by the total number of times the locks spin. The larger the *success rate*, the lower the contention level. We see that contention level increases for a certain problem size with the number of processors. This effect is more obvious when the input size is small, for example, with hundreds of vertices. For large problem size, for example, millions of vertices, there is no clear difference in contention for 2 and 16 processors. In our experiments, *success rate* is above 97% for input sizes with more than 4096 vertices, and is above 99.98% for 1M vertices, regardless the number of processors (between 2 and 16).

4.2 Spanning Tree Results

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 briefly describe the four implementations.

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

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

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.

In Figs. 4–6 we plot the performance of our spanning tree algorithms using several graph instances on Sun E4500, and in Figs. 7–9 we plot the corresponding performance using the IBM p570. Note that we use larger instances on the IBM p570 than on the Sun E4500 because of the IBM's larger main memory. In these performance results, we see that **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 on both systems although the implementation of lock-free protocols and spinlocks use different hardware atomic operations on the two systems. The performance differences in these approaches is primarily due to the nondeterminism inherent in the algorithm. For example, in Fig. 5, **span-race** runs slower than **span-lockfree** or **span-spinlock**. This is due to races among processors that actually incur one more round of iteration for **span-race** to find the spanning tree.

There is some difference in the performance of **span-lock** on the two platforms. The scaling of **span-lock** on IBM p570 is better than on Sun E4500. This may be due to the different implementations of mutex locks on the two systems. The implementation of system mutex locks usually adopts a hybrid approach, that is, the lock busy waits for a while before yielding control to the operating system. Depending on the processor speed, the cost of context switch, and the application scenario, the implementation of system mutex lock chooses a judicious amount of time to busy wait. On the Sun E4500, the mutex lock implementation is not particularly friendly for the access pattern to shared objects generated by our algorithms.

4.3 MST Results

Performance results on Sun E4500 are shown in Figs. 10–12. These empirical results demonstrate that **Bor-FAL** is the fastest implementation for sparse random graphs, and **Bor-ALM** is the fastest implementation for meshes. From 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 Figs. 10–12, 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

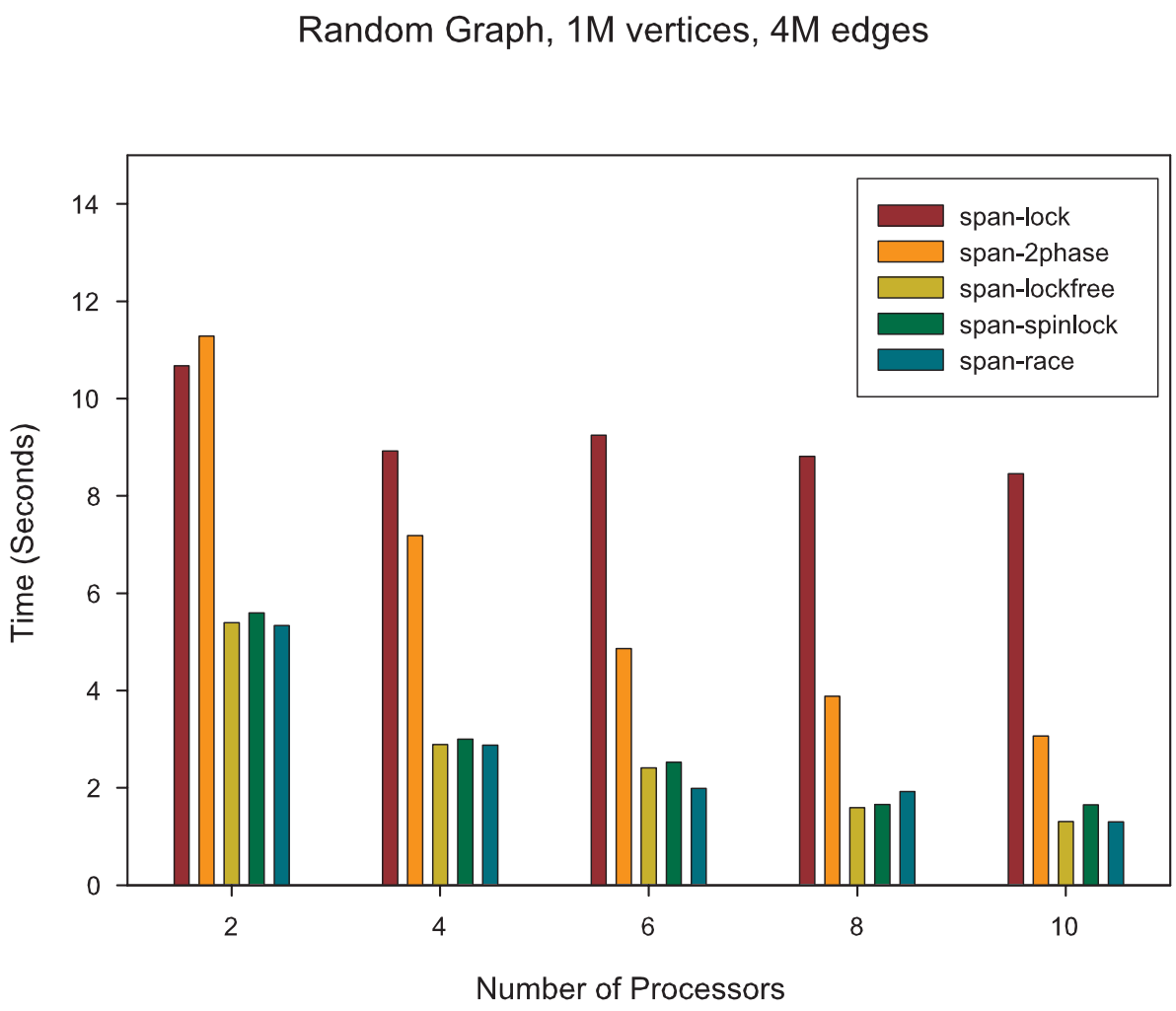

Figure 4: The performance on Sun E4500 of the spanning tree implementations on an instance of a random graph with $1M$ vertices and $4M$ edges. The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively.

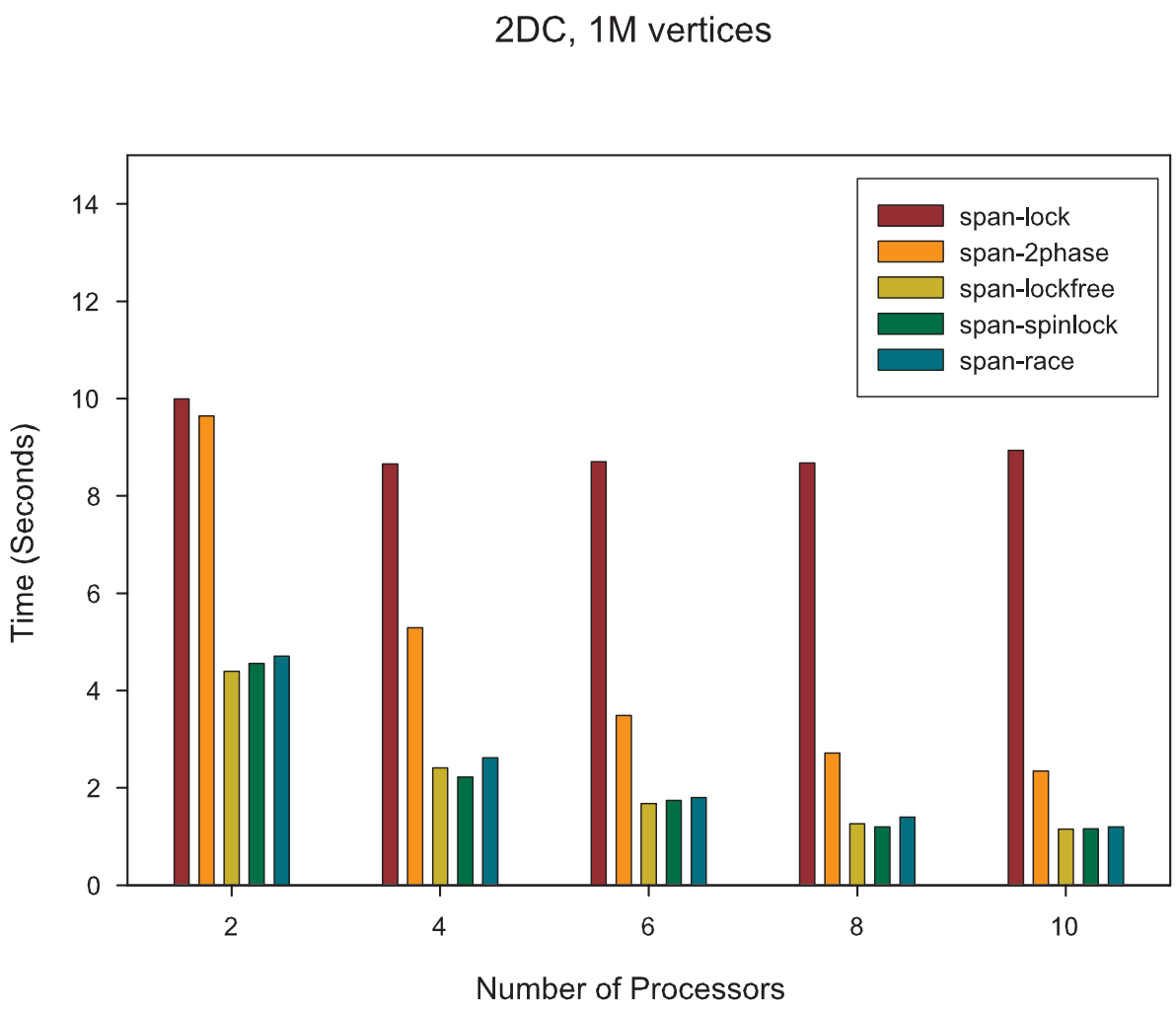

Figure 5: The performance on Sun E4500 of the spanning tree implementations on an instance of a regular, complete 2D mesh graph with $1M$ vertices. The The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively.

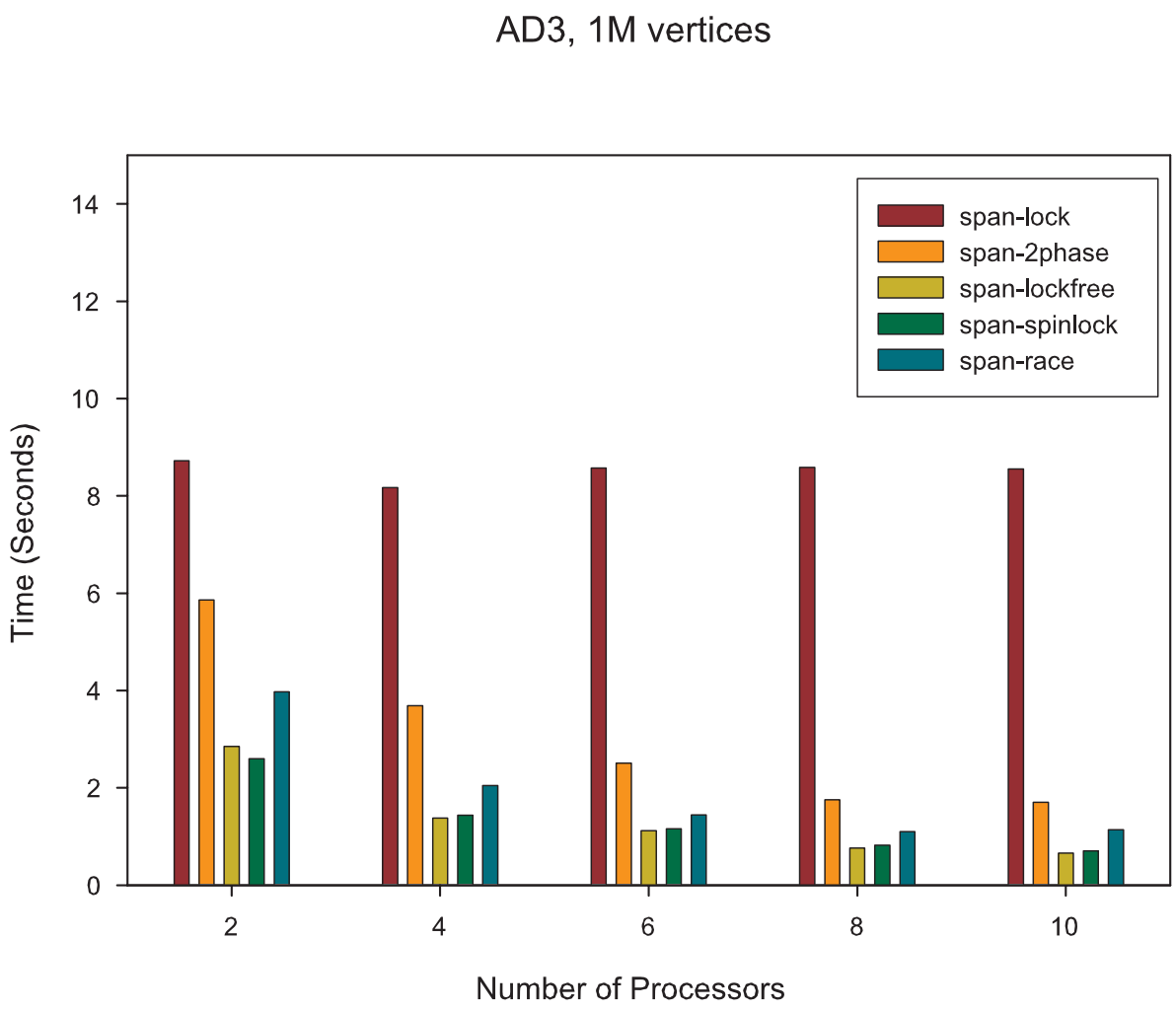

Figure 6: The performance on Sun E4500 of the spanning tree implementations on an instance of **AC3**, a geometric graph where each vertex has fixed degree $k = 3$, with $1M$ vertices. The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively.

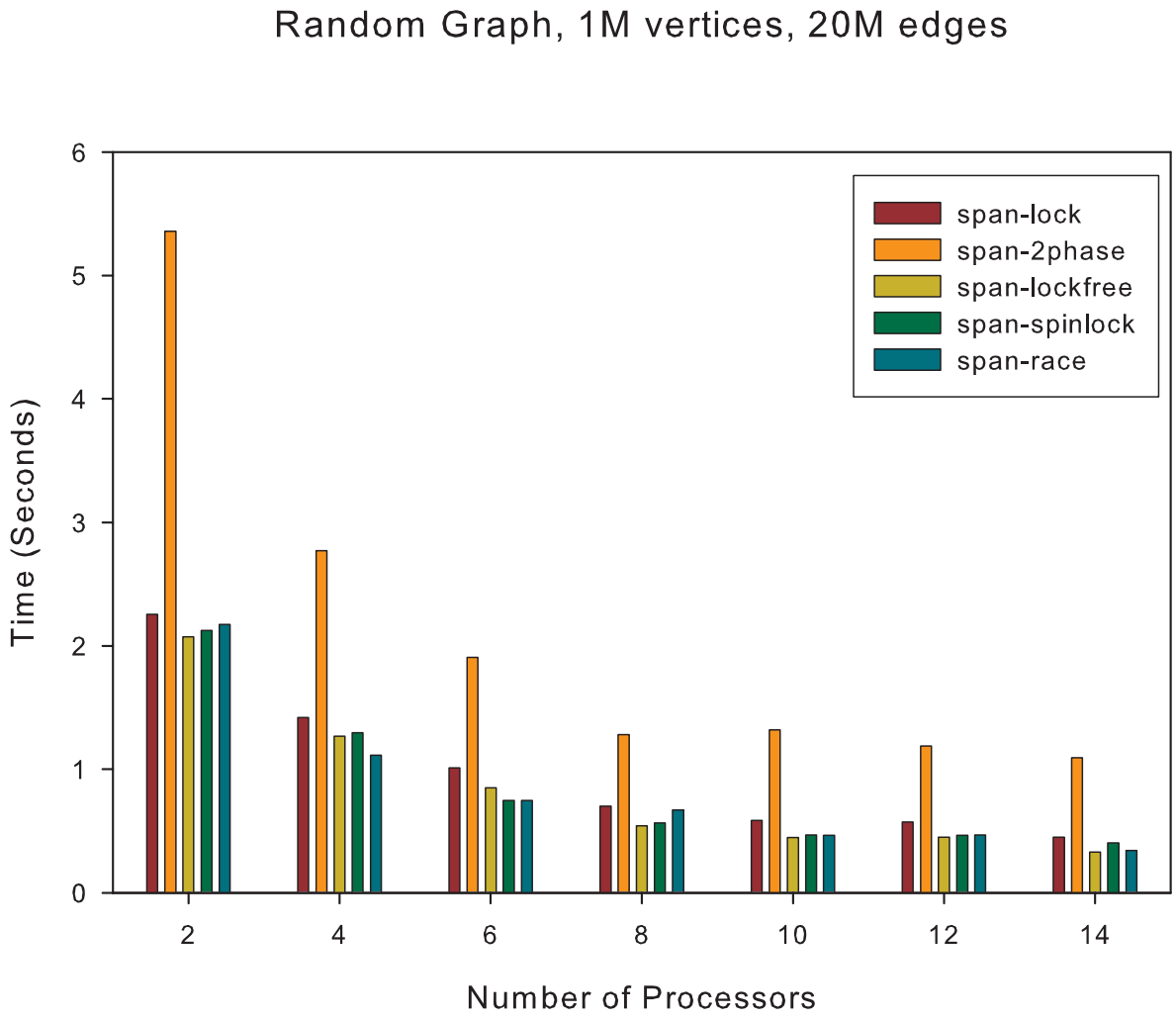

Figure 7: The performance on IBM p570 of the spanning tree implementations on an instance of random graph, with $1M$ vertices and $20M$ vertices. The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively.

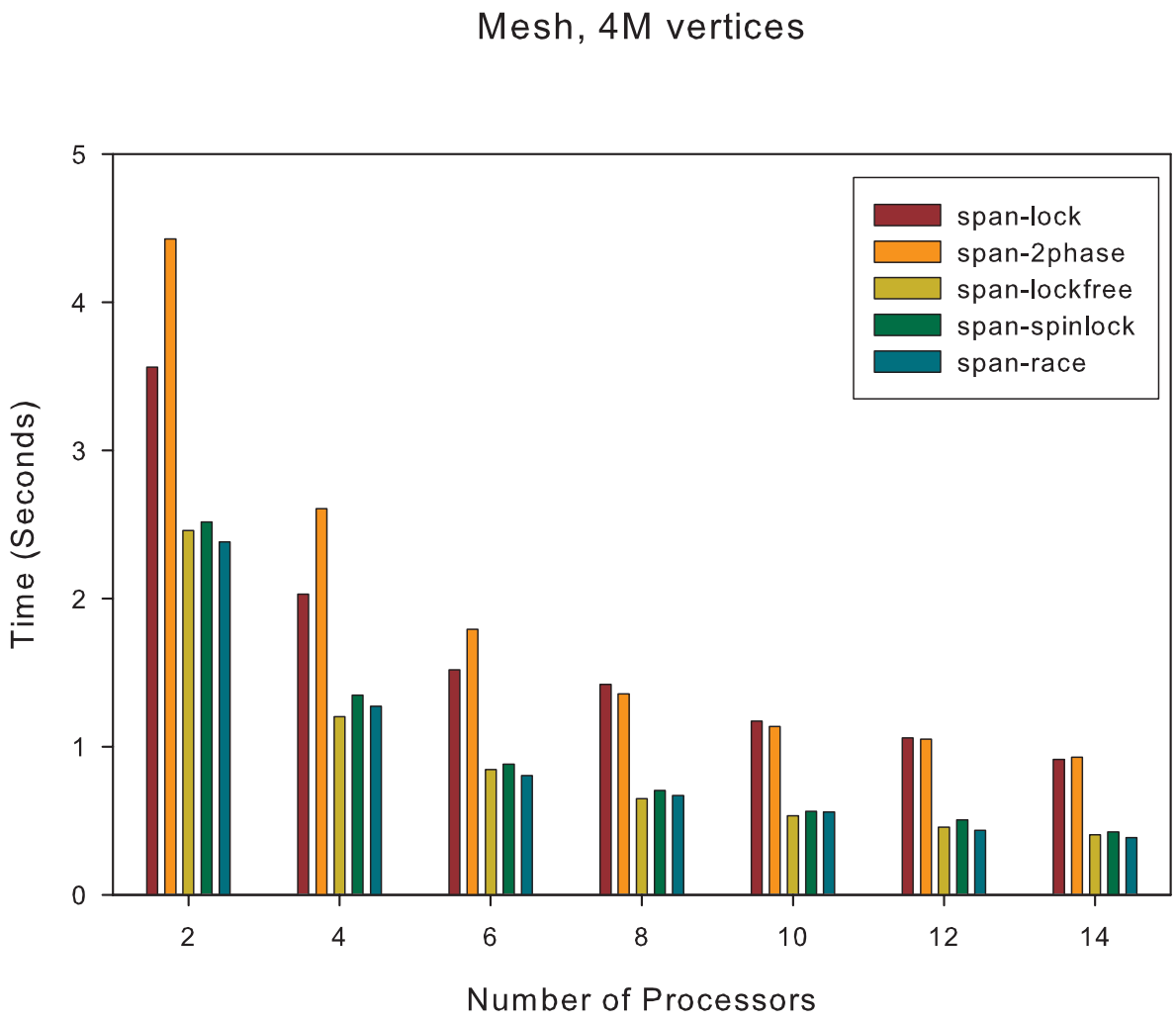

Figure 8: The performance on IBM p570 of the spanning tree implementations on an instance of **2DC**, with $4M$ vertices. The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively.

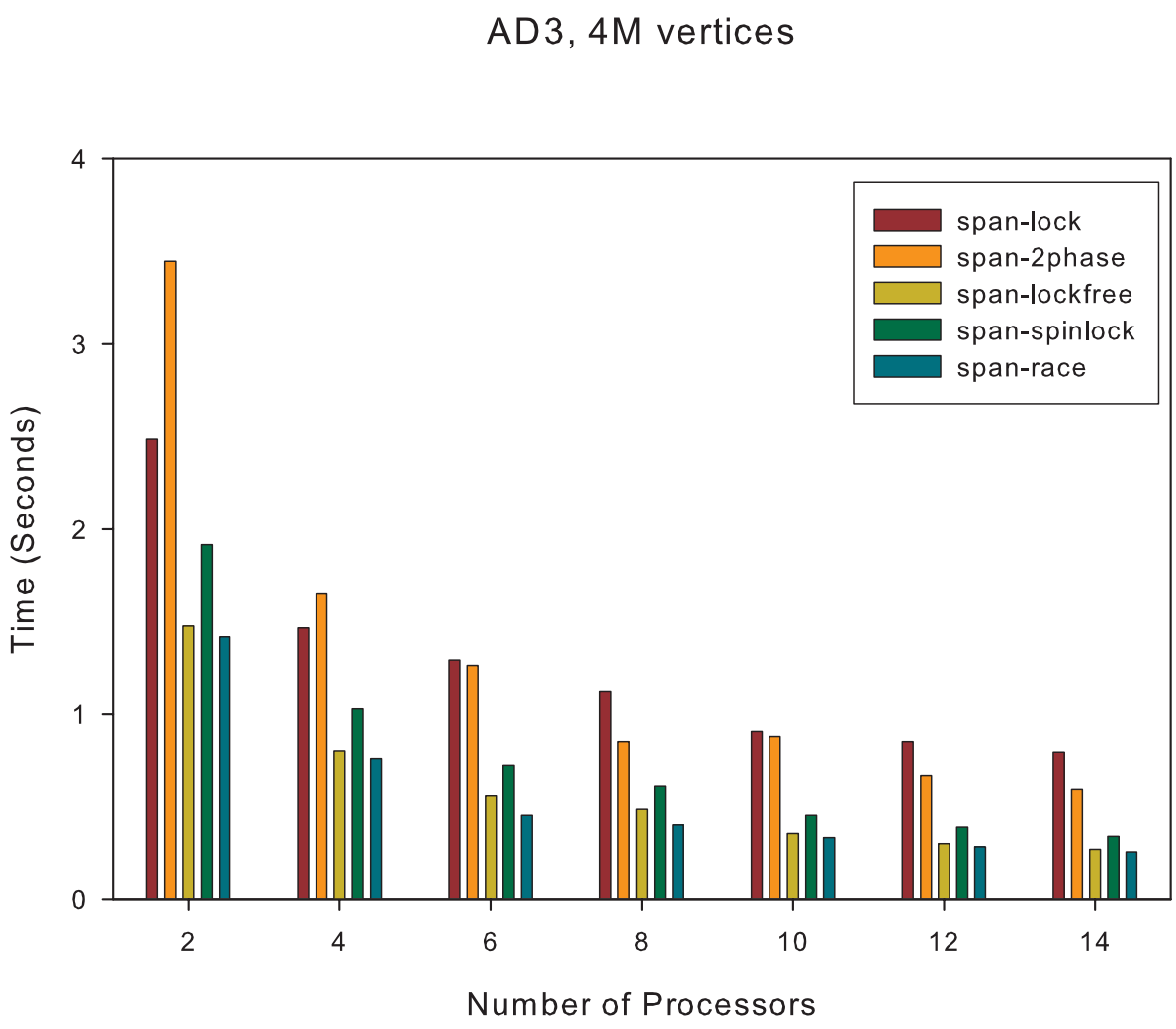

Figure 9: The performance on IBM p570 of the spanning tree implementations on an instance of **AC3**, a geometric graph where each vertex has fixed degree $k = 3$, with $4M$ vertices. The vertical bars from left to right are **span-lock**, **span-2phase**, **span-lockfree**, **span-spinlock**, and **span-race**, respectively.

as a horizontal line for each input graph. For all the input graphs shown in Figs. 10–12, **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. These performance results demonstrate 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.

Performance results on p575 are shown in Figs. 13–14. Compared with results on Sun E4500, again **Bor-lock** scales better on IBM p570, yet there is still a big gap between **Bor-lock** and **Bor-spinlock** due to the economic memory usage of spinlock and its simple implementation.

5 Conclusions

In this paper we present novel applications of lock-free protocols and fine-grained mutual exclusion locks 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 protocols and fine-grained locks to broader classes of irregular algorithms, for example, algorithms for combinatorial optimization.

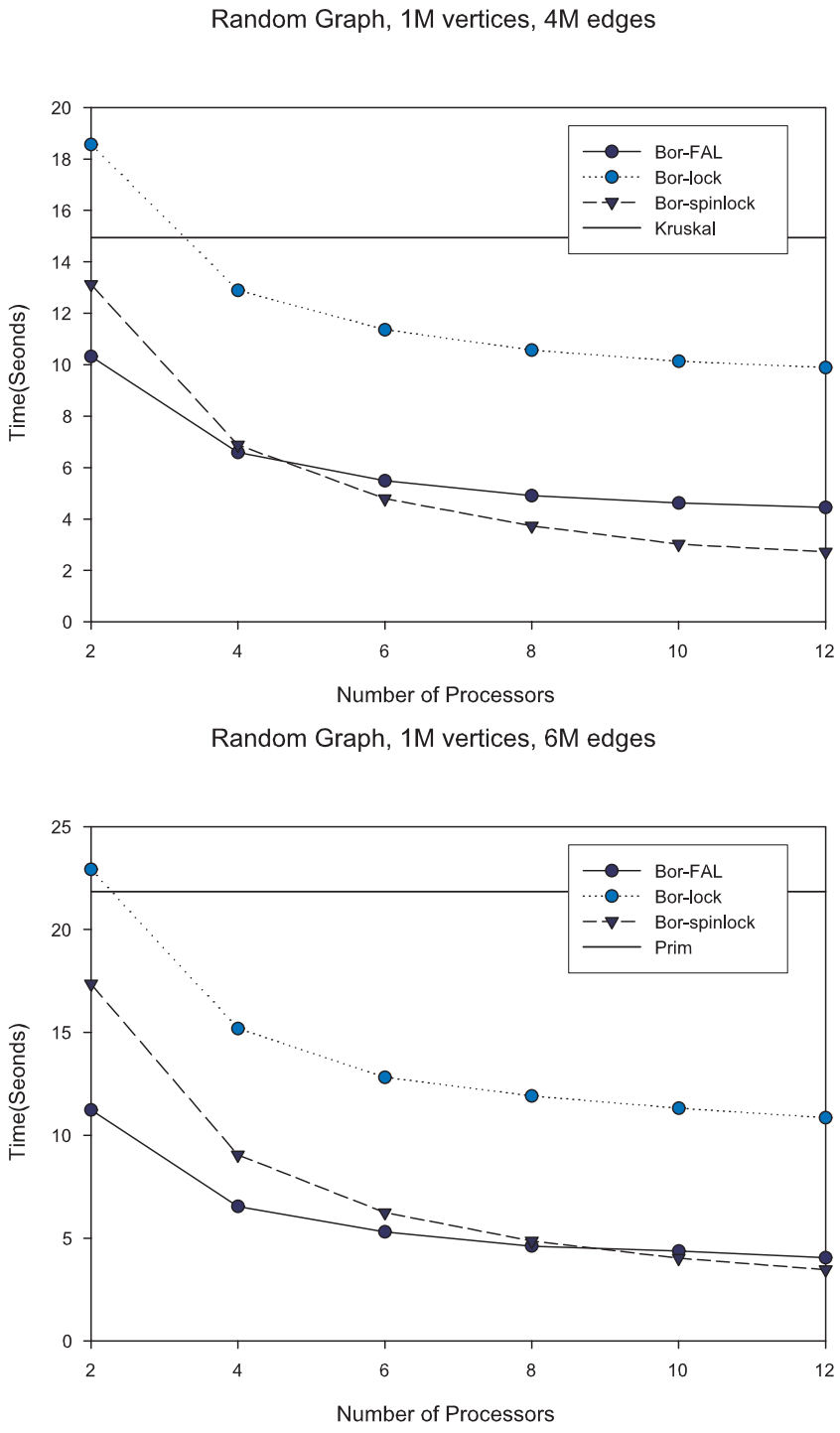

Figure 10: Comparison of the performance of **Bor-spinlock** on the Sun E4500 against the previous implementations on random graphs with $1M$ vertices and $4M$ and $6M$ edges on the top and bottom, respectively. The horizontal line in each graph shows the execution time of the best sequential implementation.

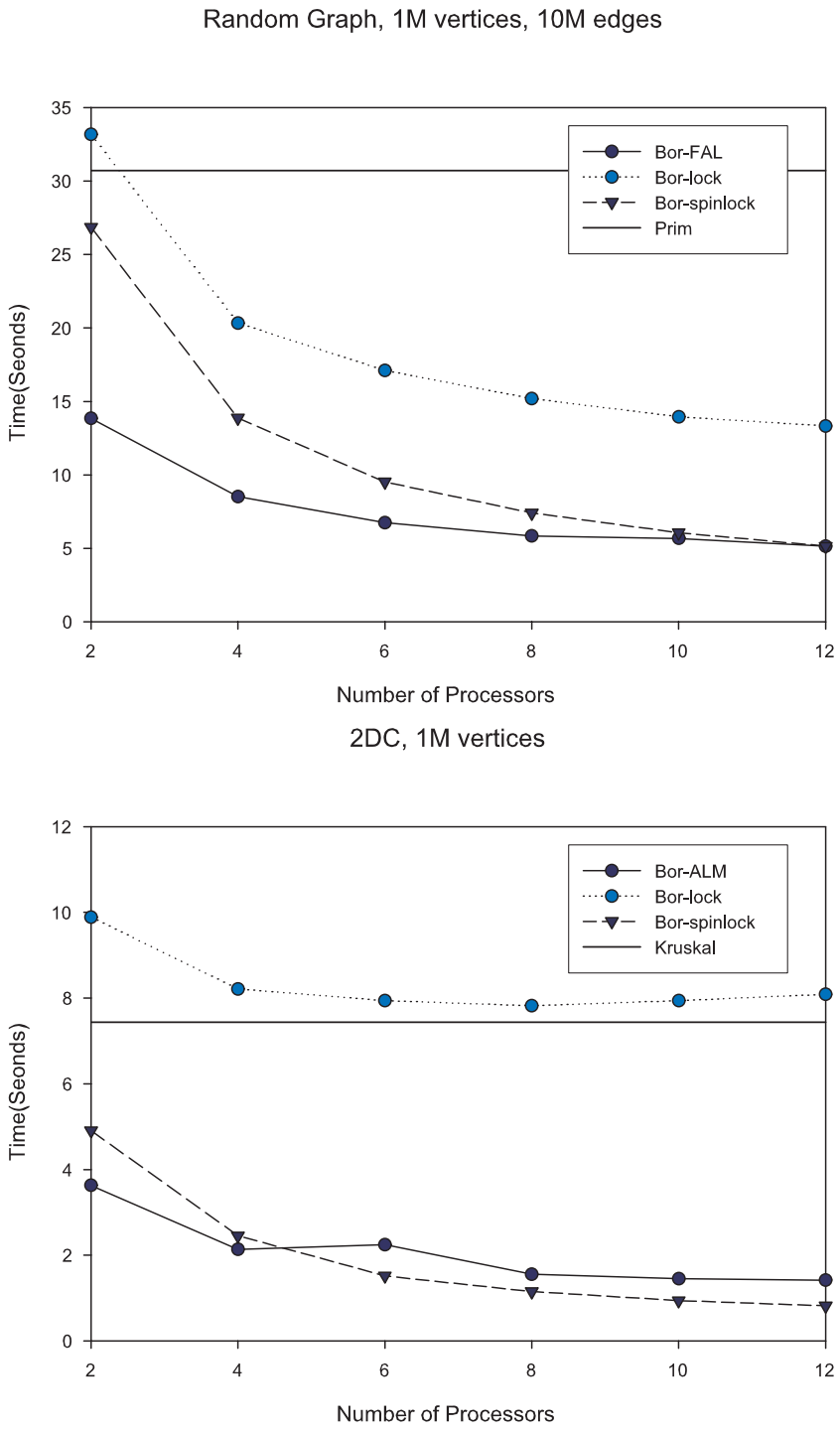

Figure 11: Comparison of the performance of **Bor-spinlock** on the Sun E4500 against the previous implementations on a random graph with $1M$ vertices and $10M$ edges (top) and on a regular 2D mesh (bottom). The horizontal line in each graph shows the execution time of the best sequential implementation.

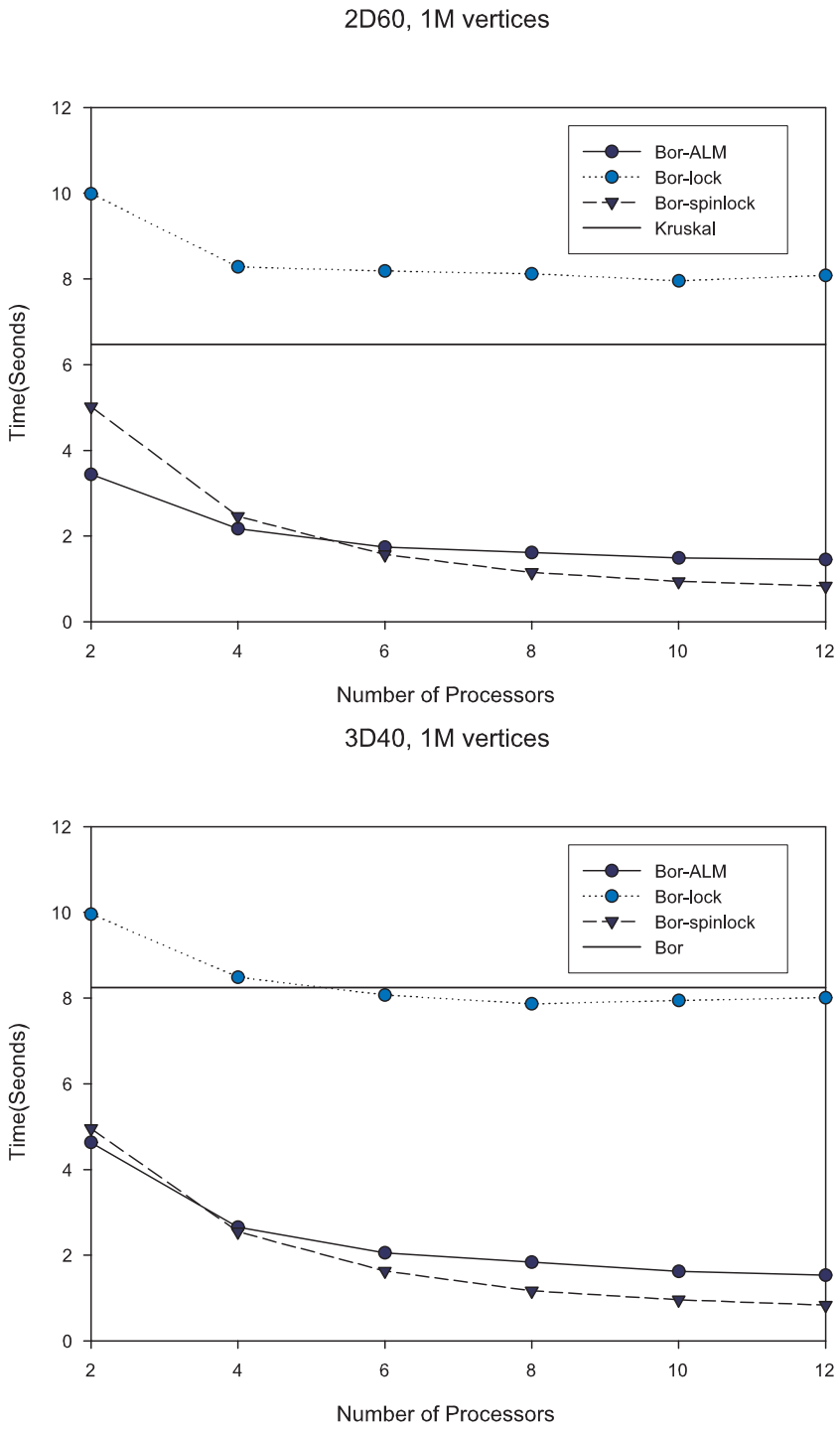

Figure 12: Comparison of the performance of **Bor-spinlock** on the Sun E4500 against the previous implementations on irregular meshes with $1M$ vertices: **2D60** (top) and **3D40** (bottom). The horizontal line in each graph shows the execution time of the best sequential implementation.

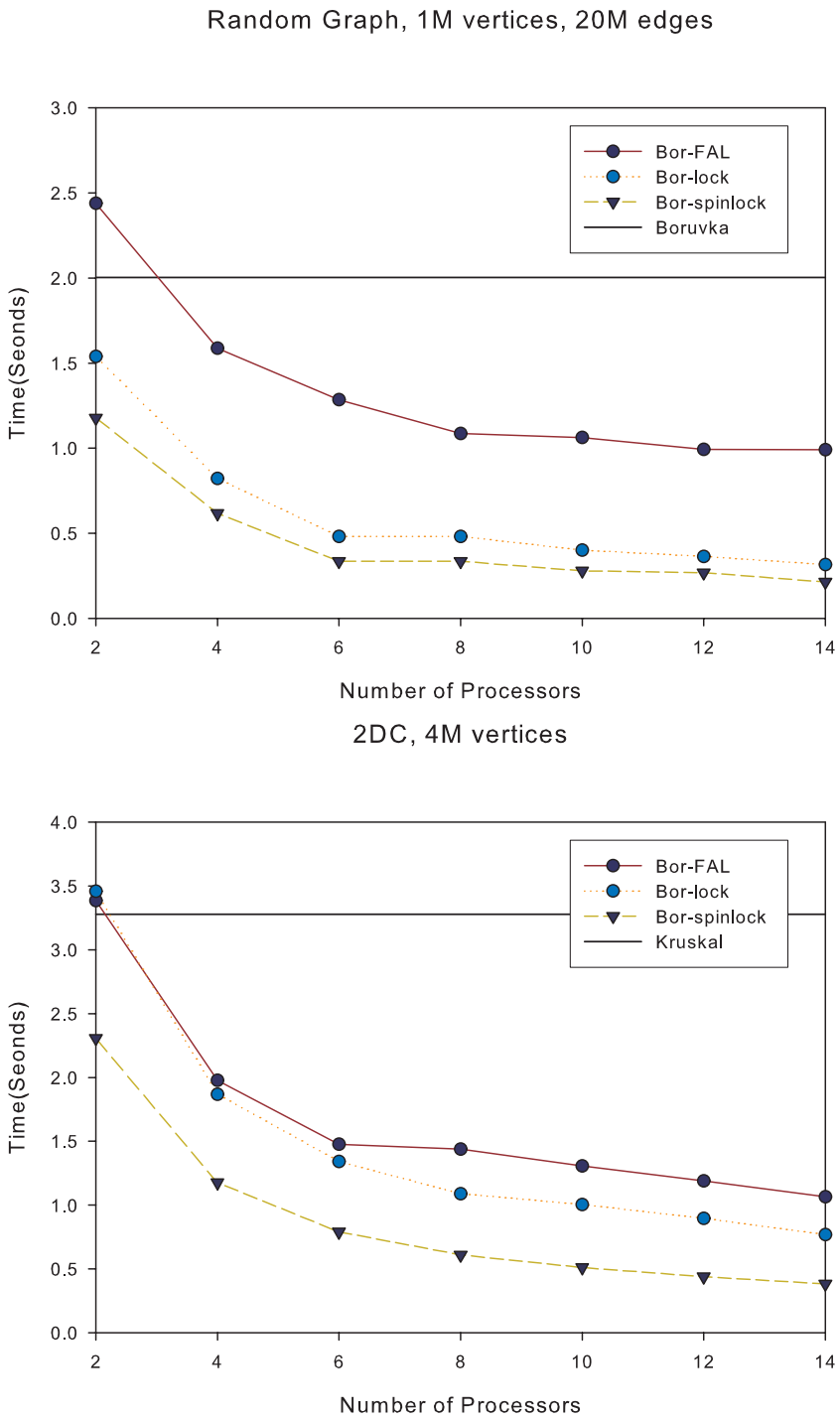

Figure 13: Comparison of the performance of **Bor-spinlock** on the IBM p570 against the previous implementations on irregular meshes with $1M$ vertices: **random** (top) and **2DC** (bottom). The horizontal line in each graph shows the execution time of the best sequential implementation.

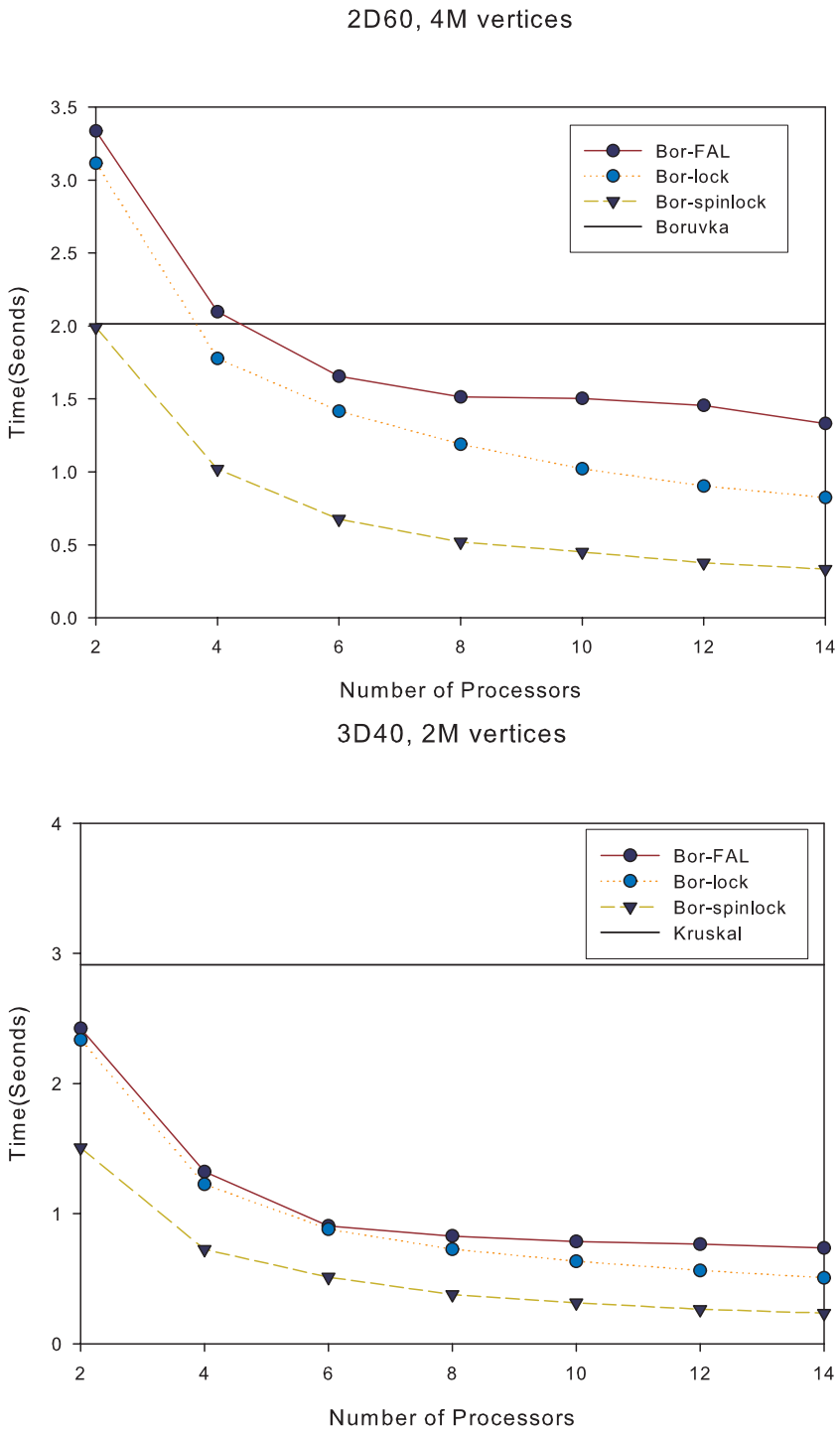

Figure 14: Comparison of the performance of **Bor-spinlock** on the IBM p570 against the previous implementations on irregular meshes with $1M$ vertices: **2D60** (top) and **3D40** (bottom). The horizontal line in each graph shows the execution time of the best sequential implementation.

References

[1] J. Alemany and E.W. Felton. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proc. 11th ACM Symp, on Principles of Distributed Computing*, pages 125–134, Vancouver, Canada, August 1992.

[2] L. An, Q.S. Xiang, and S. Chavez. A fast implementation of the minimum spanning tree method for phase unwrapping. *IEEE Trans. Med. Imaging*, 19(8):805–808, 2000.

[3] R.J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. 23rd Ann. ACM Symp. on Theory of Computing (STOC)*, pages 370–380, New Orleans, LA, May 1991.

[4] 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, Crete, Greece, July 1990.

[5] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, 1994.

[6] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, April 2004.

[7] 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)*, Santa Fe, NM, April 2004.

[8] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994–1006, 2005.

[9] D.A. Bader, A.K. Illendula, B. M.E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[10] D.A. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V.K. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, December 2002. Springer-Verlag.

[11] G. Barnes. Wait-free algorithms for heaps. Technical Report TR-94-12-07, University of Washington, Seattle, WA, 1994.

[12] M. Brinkhuis, G.A. Meijer, P.J. van Diest, L.T. Schuurmans, and J.P. Baak. Minimum spanning tree analysis in advanced ovarian carcinoma. *Anal. Quant. Cytol. Histol.*, 19(3):194–201, 1997.

[13] C. Chen and S. Morris. Visualizing evolving networks: Minimum spanning trees versus pathfinder networks. In *IEEE Symp. on Information Visualization*, Seattle, WA, October 2003. to appear.

[14] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 86–97, Vancouver, Canada, August 1987.

[15] 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.

[16] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[17] R. Cole, P.N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proc. 8th Ann. Symp. Parallel Algorithms and Architectures (SPAA-96)*, pages 243–250, Newport, RI, June 1996. ACM.

[18] R. Cole, P.N. Klein, and R.E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 11–15, Newport, RI, June 1994. ACM.

[19] 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, Santa Fe, NM, June 1989.

[20] G. Cong and D. A. Bader. The Euler tour technique and parallel rooted spanning tree. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, pages 448–457, Montreal, Canada, August 2004.

[21] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th Symp. Principles and Practice of Parallel Programming*, pages 1–12. ACM SIGPLAN, May 1993.

[22] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[23] J.C. Dore, J. Gilbert, E. Bignon, A. Crastes de Paulet, T. Ojasoo, M. Pons, J.P. Raynaud, and J.F. Miquel. Multivariate analysis by the minimum spanning tree method of the structural determinants of diphenylethylenes and triphenylacrylonitriles implicated in estrogen receptor binding, protein kinase C activity, and MCF7 cell proliferation. *J. Med. Chem.*, 35(3):573–583, 1992.

[24] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[25] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[26] K. Fraser. *Practical lock-freedom.* PhD thesis, King's College, University of Cambridge, United Kingdom, September 2003.

[27] 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.

[28] 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 Transactions on Computers*, C-32(2):175–189, 1984.

[29] 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.

[30] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Int'l Symp. on Computer Architecture*, pages 289–300, San Diego, CA, May 1993.

[31] M.P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[32] M.P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[33] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *Proc. 14th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 13–26, Munich, West Germany, January 1987.

[34] 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.

[35] IBM. *Assembler Language Reference, AIX 4.3 books*, 1 edition, 1997.

[36] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, New York, 1992.

[37] K. Kayser, S.D. Jacinto, G. Bohm, P. Frits, W.P. Kunze, A. Nehrlich, and H.J. Gabius. Application of computer-assisted morphometry to the analysis of prenatal development of human lung. *Anat. Histol. Embryol.*, 26(2):135–139, 1997.

[38] K. Kayser, H. Stute, and M. Tacke. Minimum spanning tree, integrated optical density and lymph node metastasis in bronchial carcinoma. *Anal. Cell Pathol.*, 5(4):225–234, 1993.

[39] V.E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[40] 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.

[41] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proc. 13th Ann. ACM Symp. on Principles of Distributed Computing*, pages 130–140, Los Angeles, CA, August 1994.

[42] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[43] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[44] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 211–220, Austin, TX, March 1988.

[45] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.

[46] H. Massalin and C. Pu. Threads and input/output in the synthesis kernel. In *Proc. 12th ACM Symp. on Operating Systems Principles (SOSP)*, pages 191–201, Litchfield Park, AZ, December 1989.

[47] M. Matos, B.N. Raby, J.M. Zahm, M. Polette, P. Birembaut, and N. Bonnet. Cell migration and proliferation are not discriminatory factors in the in vitro sociologic behavior of bronchial epithelial cell lines. *Cell Motility and the Cytoskeleton*, 53(1):53–65, 2002.

[48] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M. Srivastava. Coverage problems in wireless ad-hoc sensor networks. In *Proc. INFOCOM '01*, pages 1380–1387, Anchorage, AK, April 2001. IEEE Press.

[49] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[50] G. L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Manuscript, UC Berkeley, MSRI, January 1986.

[51] 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.

[52] V. Olman, D. Xu, and Y. Xu. Identification of regulatory binding sites using minimum spanning trees. In *Proc. 8th Pacific Symp. Biocomputing (PSB 2003)*, pages 327–338, Hawaii, 2003. World Scientific Pub.

[53] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002.

[54] C.K. Poon and V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *Proc. 8th Int'l Symp. Algorithms and Computation (ISAAC'97)*, volume 1350 of *Lecture Notes in Computer Science*, pages 212–222. Springer-Verlag, 1997.

[55] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th Ann. ACM Symp. on Principles of Distributed Computing*, pages 204–213, Ottowa, Canada, August 1995.

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

[57] R.E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

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

[59] Y.-C. Tseng, T.T.-Y. Juang, and M.-C. Du. Building a multicasting tree in a high-speed network. *IEEE Concurrency*, 6(4):57–67, 1998.

[60] 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, Crete, Greece, September 2001.

[61] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[62] J. Valois. *Lock-free data structures.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, May 1995.

[63] J.D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th Ann. ACM Symp. on Principles of Distributed Computing*, pages 214–222, Ottowa, Canada, August 1995.

[64] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20(5):235–240, 1985.

[65] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism. In *Proc. 10th Ann. Symp. Parallel Algorithms and Architectures (SPAA-98)*, pages 140–151, Puerto Vallarta, Mexico, June 1998. ACM.

[66] D.L. Weaver and T. Germond, editors. *The SPARC architecture manual, version 9.* Prentice Hall, 1994.

[67] S.Q. Zheng, J.S. Lim, and S.S. Iyengar. Routing using implicit connection graphs. In *9th Int'l Conf. on VLSI Design: VLSI in Mobile Communication*, Bangalore, India, January 1996. IEEE Computer Society Press.