A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs)

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

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

February 25, 2006

Abstract

The ability to provide uniform shared-memory access to a significant number of processors in a single SMP node brings us much closer to the ideal PRAM parallel computer. Many PRAM algorithms can be adapted to SMPs with few modifications. Yet there are few studies that deal with the implementation and performance issues of running PRAM-style algorithms on SMPs. Our study in this paper focuses on implementing parallel spanning tree algorithms on SMPs. Spanning tree is an important problem in the sense that it is the building block for many other parallel graph algorithms and also because it is representative of a large class of irregular combinatorial problems that have simple and efficient sequential implementations and fast PRAM algorithms, but these irregular problems often have no known efficient parallel implementations. Experimental studies have been conducted on related problems (minimum spanning tree and connected components) using parallel computers, but only achieved reasonable speedup on regular graph topologies that can be implicitly partitioned with good locality features or on very dense graphs with limited numbers of vertices. In this paper we present a new randomized algorithm and implementation with superior performance that *for the first time* achieves parallel speedup on arbitrary graphs (both regular and irregular topologies) when compared with the best sequential implementation for finding a spanning tree. This new algorithm uses several techniques to give an expected running time that scales linearly with the number p of processors for suitably large inputs ($n > p^2$). As the spanning tree problem is notoriously hard for any parallel implementation to achieve reasonable speedup, our study may shed new light on implementing PRAM algorithms for shared-memory parallel computers. The main results of this paper are

1. A new and practical spanning tree algorithm for symmetric multiprocessors that exhibits parallel speedups on graphs with regular and irregular topologies; and
2. An experimental study of parallel spanning tree algorithms that reveals the superior performance of our new approach compared with the previous algorithms.

The source code for these algorithms is freely-available from our web site `hpc.ece.unm.edu`.

*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

Finding a spanning tree of a graph is an important building block for many graph algorithms, for example, biconnected components and ear decomposition [33], and can be used in graph planarity testing [29]. The best sequential algorithm for finding a spanning tree of a graph $G = (V, E)$ where $n = |V|$ and $m = |E|$ uses depth- or breadth-first graph traversal and runs in $O(m + n)$. The implementation of the sequential algorithms are very efficient (linear time with a very small hidden constant), and the only data structure used is a stack or queue which has good locality features. However, graph traversal using depth-first search (DFS) is inherently sequential and known not to parallelize efficiently [38]. Thus, the previous approaches for parallel spanning tree algorithms use novel techniques other than traversal that are conducive to parallelism and have polylogarithmic time complexities. In practice, none of these parallel algorithms has shown significant parallel speedup over the best sequential algorithm for sparse, irregular graphs, because the theoretic models do not realistically capture the cost for communication on current parallel machines (e.g., [1, 10, 11, 13, 15, 18, 19, 22, 24, 26, 27, 28, 31, 35, 37, 41]), the algorithm is too complex for implementation (e.g., [11, 18]), or there are large constants hidden in the asymptotic notation that could not be overcome by a parallel implementation (e.g., [12, 14, 17, 23, 30]). In our studies, we consider a graph as sparse when $m = O(n \log n)$.

Symmetric multiprocessor (SMP) architectures, in which several processors operate in a true, hardware-based, shared-memory environment are becoming commonplace. Indeed, most of the new high-performance computers are clusters of SMPs having from 2 to over 100 processors per node. The ability to provide uniform-memory-access (UMA) shared-memory for a significant number of processors brings us much closer to the ideal parallel computer envisioned over 20 years ago by theoreticians, the *Parallel Random Access Machine (PRAM)* (see [25, 39]) and thus may enable us at last to take advantage of 20 years of research in PRAM algorithms for various irregular computations (such as spanning tree and other graph algorithms). Moreover, as supercomputers increasingly use SMP clusters, SMP computations will play a significant role in supercomputing.

While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work—synchronization cannot be taken for granted, memory bandwidth is limited, and performance requires a high degree of locality. The significant feature of SMPs is that they provide much faster access to their shared-memory than an equivalent message-based architecture. Even the largest SMP to date, the 106-processor Sun Fire Enterprise 15000 (E15K) [8, 9], has a worst-case memory access time of 450ns (from any processor to any location within its 576GB memory); in contrast, the latency for access to the memory of another processor in a distributed-memory architecture is measured in tens of μs. In other words, message-based architectures are two orders of magnitude slower than the largest SMPs in terms of their worst-case memory access times.

The Sun E15K uses a combination of data crossbar switches, multiple snooping buses, and sophisticated cache handling to achieve UMA across the entire memory. Of course, there remains a large difference between the access time for an element in the local processor cache (around 10ns) and that for an element that must be obtained from memory (at most 450ns)—and that difference increases as the number of processors

Model (PRAM)	Authors	Time	Work
priority CRCW	Shiloach, Vishkin[41]	$O(\log n)$	$O((m+n)\log n)$
	Awerbuch, Shiloach [1]	$O(\log n)$	$O((m+n)\log n)$
arbitrary CRCW	Cole, Vishkin [13]	$O(\log n)$	$O((m+n)\alpha(m,n))$
	Iwana, Kambayashi [24]	$O(\log n)$	$O((m+n)\alpha(m,n))$
CREW	Hirschberg *et al.* [22]	$O\left(\log^2 n\right)$	$O(n^2 \log n)$
	Chin *et al.* [10]	$O\left(\log^2 n\right)$	$O(n^2)$
	Han, Wagner [19]	$O\left(\log^2 n\right)$	$O((m+n\log n)\log n)$
	Johnson, Metaxas [26]	$O\left(\log^{3/2} n\right)$	$O\left((m+n)\log^{3/2} n\right)$
EREW	Nash, Maheshwari [35]	$O\left(\log^2 n\right)$	$O(n^2)$
	Phillips [37]	$O\left(\log^2 n\right)$	$O((m+n)\log n)$
	Kruskal *et al.* [31]	$O\left(\log^2 n\right)$	$O((m+n)\log n)$
	Johnson, Metaxas [27]	$O\left(\log^{3/2} n\right)$	$O\left((m+n)\log^{3/2} n\right)$
	Chong, Lam [11]	$O(\log n \log\log n)$	$O((m+n)\log n \log\log n)$

Table 1: Deterministic spanning tree algorithms for CRCW, CREW and EREW PRAMs. α is the inverse Ackermann's function.

increases, so that cache-aware implementations are even more important on large SMPs than on single workstations.

The main results of this paper are 1) a new and practical spanning tree algorithm for symmetric multiprocessors that exhibits parallel speedups on graphs with regular and irregular topologies; and 2) an experimental study of parallel spanning tree algorithms that reveals the superior performance of our new approach compared with the previous algorithms. For realistic problem sizes ($n \gg p^2$), the expected running time for our new SMP spanning tree algorithm on a graph with n vertices and m edges is given by $T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle \leq \left\langle O\left(\frac{n+m}{p}\right)\ ;\ O\left(\frac{n+m}{p}\right)\ ;\ 2\right\rangle$ where p is the number of processors, using the SMP complexity model described in Section 3. A preliminary version of this paper appeared in [2]. In Sections 1.1 and 1.2 we survey the theoretic and experimental literature, respectively, for prior results in parallel spanning tree and related research.

1.1 Parallel Algorithms for Spanning Tree

For a sparse graph $G = (V, E)$ where $n = |V|$ and $m = |E|$, various deterministic and randomized techniques have been given for solving the spanning tree problem on PRAM models. Known deterministic results are in Table 1. Some of the algorithms are related, for example: Iwana and Kambayashi's algorithm improves the Cole and Vishkin algorithm by removing the expander graph so that the hidden constant in the asymptotic notation becomes smaller; Chin *et al.* improve Hirschberg *et al.*'s algorithm by exploiting the adjacency matrix as the representing data structure; and Nash and Maheshwari's algorithm improves Hirschberg *et al.*'s algorithm by building data structures to eliminate the concurrent writes.

Gazit [15] and Halperin and Zwick [18] have designed optimal randomized approaches for parallel spanning tree that run in $O(\log n)$ time with high probability on the CRCW and EREW PRAM, respectively. The algorithm of Halperin and Zwick [18] combines techniques from several previous algorithms; it borrows the *maximum-hooking* method from Chong and Lam [11] to resolve possible grafting conflicts, complicated *growth control* method from Johnson and Metaxas [26, 27] which is the key technique for them to achieve an $O\left(\log^{3/2} n\right)$ algorithm, and other techniques from Gazit [15] and Karger, Klein, and Tarjan [28].

1.2 Related Experimental Studies

As we described in the previous section, the research community has produced a rich collection of theoretic deterministic and randomized spanning tree algorithms. Yet for implementations and experimental studies, although several fast PRAM spanning tree algorithms exist, to our knowledge there is no parallel implementation of spanning tree (or the related problems such as minimum spanning tree and connected components that produce a spanning tree) that achieves significant parallel speedup on sparse, irregular graphs when compared against the best sequential implementation. In our study we carefully chose several known PRAM algorithms and implemented them for shared-memory (using appropriate optimizations described by Greiner [17], Chung and Condon [12], Krishnamurthy *et al.* [30], and Hsu *et al.* [23]), and compared these with our new randomized approach. Our new algorithm to our knowledge is the first to achieve any reasonable parallel speedup for both regular and irregular graphs.

Greiner [17] implemented several connected components algorithms (Shiloach-Vishkin, Awerbuch-Shiloach, "random-mating" based on the work of Reif [40] and Phillips [37], and a hybrid of the previous three) using NESL on the Cray Y-MP/C90 and TMC CM-2. On random graphs Greiner reports a maximum speedup of 3.5 using the hybrid algorithm when compared with a depth-first search on a DEC Alpha processor. Hsu, Ramachandran, and Dean [23] also implemented several parallel algorithms for connected components. They report that their parallel code runs 30 times slower on a MasPar MP-1 than Greiner's results on the Cray, but Hsu *et al.*'s implementation uses one-fourth of the total memory used by Greiner's hybrid approach. Krishnamurthy *et al.* [30] implemented a connected components algorithm (based on Shiloach-Vishkin [41]) for distributed memory machines. Their code achieved a speedup of 20 using a 32-processor TMC CM-5 on graphs with underlying 2D and 3D regular mesh topologies, but virtually no speedup on sparse random graphs. Goddard, Kumar, and Prins [16] implemented a connected components algorithm (motived by Shiloach-Vishkin) for a mesh-connected SIMD parallel computer, the 8192-processor MasPar MP-1. They achieve a maximum parallel speedup of less than two on a random graph with 4096 vertices and about one-million edges. For a random graph with 4096 vertices and fewer than a half-million edges, the parallel implementation was slower than the sequential code. Chung and Condon [12] implemented a parallel minimum spanning tree (MST) algorithm based on Borůvka's algorithm. On a 16-processor CM-5, for geometric graphs with 32,000 vertices and average degree 9 and graphs with fewer vertices but higher average degree, their code achieved a parallel speedup of about 4, on 16-processors, over the sequential Borůvka's algorithm, which was 2–3 times slower than their sequential Kruskal algorithm. Dehne and Götz [14] studied practical parallel al-

gorithms for MST using the BSP model. They implemented a dense Borůvka parallel algorithm, on a 16-processor Parsytec CC-48, that works well for sufficiently dense input graphs. Using a fixed-sized input graph with 1,000 vertices and 400,000 edges, their code achieved a maximum speedup of 6.1 using 16 processors for a random dense graph. Their algorithm is not suitable for sparse graphs.

Section 2 further details the parallel algorithms we designed and implemented. The shared-memory analysis of these algorithms is given in Section 3. In Section 4 we detail the experimental study, describe the input data sets and testing environment, and present the experimental results. Finally, Section 5 provides our conclusions and future work.

2 Parallel Spanning Tree Algorithms for SMPs

Here we present the three parallel spanning tree algorithms we have implemented. Based on the asymptotic complexities of the algorithms, programming complexity, and constant factors hidden in the asymptotic notation, we choose two representative PRAM algorithms to implement for SMPs: the Shiloach-Vishkin (SV) and the Hirschberg-Chandra-Sarwate (HCS) algorithms, using appropriate optimizations suggested by [12, 17, 23, 30]. Through the experience we gained by implementing these two algorithms, we developed a new randomized algorithm with superior performance in all of our experiments.

2.1 The Shiloach-Vishkin Algorithm

The Shiloach-Vishkin algorithm (SV) is in fact a connected-components algorithm [1, 41]. This algorithm is representative of several connectivity algorithms in that it adapts the widely-used graft-and-shortcut approach. Through carefully designed grafting schemes, the algorithm achieves complexities of $O(\log n)$ time and $O((m+n)\log n)$ work under the arbitrary CRCW PRAM model. It can be extended naturally to solve the spanning tree problem under the priority CRCW PRAM model with the same complexity bound. Yet for implementation on an SMP, the tightly-synchronized concurrent steps (read and write) are unrealistic and modification of the algorithm is necessary, as we discuss next.

The basic problem of adapting this algorithm (Alg. 1) on SMPs as a spanning tree algorithm is that it may graft a tree onto two or more different trees or onto the tree itself and produce cycles. This is allowable in the connected components algorithm as long as the connected vertices are labeled as in the same component, yet it will be an issue in the spanning tree algorithm for this may produce some false tree edges. It is in fact a race condition between processors that wish to graft a subtree rooted at one vertex onto different trees. The mismatch between the priority CRCW model and a real SMP is as follows. The original algorithm assumes that concurrent writes are arbitrated among the processors using the priority scheme: during each time step, if multiple processors write to a given memory location, at the end of the step, the memory contains the value written by the processor with the highest priority. The priority CRCW PRAM model assumes this arbitration can be performed in one time unit, yet most SMPs will require a cost to simulate this concurrent write policy.

Data : (1) A set of edges (i, j) given in an arbitrary order, and (2) a pseudoforest defined by a function D such that all the vertices in each tree belong to the same connected component.

Result: The pseudoforest obtained after (1) grafting trees onto smaller vertices of other trees, (2) grafting rooted stars onto other trees if possible, and (3) performing the pointer jumping operation on each vertex.

begin

while *true* **do**

1. Perform a grafting operation of trees onto smaller vertices of other trees as follows:

for *all* $(i, j) \in E$ *in parallel* **do**

if $(D(i) = D(D(i))$ ***and*** $D(j) < D(i))$ **then** set $D(D(i)) = D(j)$

2. Graft rooted stars onto other trees if possible, as follows:

for *all* $(i, j) \in E$ *in parallel* **do**

if *(i belongs to a star* ***and*** $D(j) \neq D(i))$ **then** set $D(D(i)) = D(j)$

3. If all the vertices are in rooted stars, then **exit**. Otherwise, perform the pointer jumping operation on each vertex as follows:

Set $D(D(i)) = D(i)$

end

Algorithm 1: Shiloach-Vishkin [41] PRAM Connected Components Algorithm.

One straightforward solution uses locks to ensure that a tree gets grafted only once. The locking approach intuitively is slow and not scalable, and our test results agree. Another approach is to always shortcut the tree to a rooted star (to avoid grafting a tree onto itself) and run an election among the processors that wish to graft the same tree before actually do the grafting. Only the winner of the election grafts the tree (to avoid grafting a tree onto multiple other trees). This approach is also used by other researchers [24, 17] to handle the race conditions in their spanning tree algorithms. The running time of the algorithm is now $O\left(\log^2 n\right)$; the additional $\log n$ factor comes from shortcutting (pointer jumping). Optimizations are possible for the election approach. For example, step 2 in Alg. 1 could be removed because now all the grafting can be done in step 1, and we could periodically shrink the edge list to eliminate those edges that have been used so that we do not need to scan the entire edge list each iteration. This approach is generally faster than the locking scheme, yet it also has the following major slow down factors:

1. Although the election procedure does not asymptotically affect the running time of the algorithm, it increases the hidden constant factor. Now we literally run the grafting phase of the Shiloach-Vishkin algorithm twice.
2. The SMP processors must compete for writing to the same memory location to emulate concurrent writes. Note also that with more processors available, the competition can potentially cause memory congestion if many of the processors write to the same memory location when trying to graft the same subtree.

SV is sensitive to the labeling of vertices. For the same graph topology, different labeling of vertices may incur different numbers of iterations to terminate the algorithm. For the best case, one iteration of the algorithm may be sufficient, and the running time of the algorithm will be $O(\log n)$. Whereas for an arbitrary labeling of the same graph, up to $\log n$ iterations will be needed. We expect to see similar behaviors for the class of algorithms that use the "grafting and short-cutting" approach.

2.2 The Hirschberg-Chandra-Sarwate Algorithm

The Hirschberg-Chandra-Sarwate algorithm [22] (HCS) is one of the earliest parallel graph connectivity algorithms and has $O\left(\log^2 n\right)$ time, $O(n^2 \log n)$ work complexities on CREW PRAM. The simplicity of the HCS parallel algorithm (unlike many later variants of parallel spanning tree) and its use of exclusive write make it attractive for implementation. Although we can emulate PRAM models (e.g., CRCW, CREW and EREW) on SMPs, exclusive read is perhaps too restrictive, while concurrent write incurs contention and serialization on SMPs. We expect a CREW PRAM algorithm can be more naturally emulated on the currently available SMPs. Similar to the SV algorithm, HCS is a connected-components algorithm that requires modification to transform it into a spanning tree algorithm. Our modified HCS algorithm for spanning tree results in similar complexities and running time as that of SV when implemented on an SMP, and hence, we leave it out of further discussion.

2.3 A New Spanning Tree Algorithm For SMPs

Our new parallel spanning tree algorithm for shared-memory multiprocessors has two main steps: 1) generating a stub spanning tree, and 2) performing work-stealing graph traversal. The overall strategy is first to generate a small stub spanning tree with one processor, and then let each processor start from vertices in the stub tree and traverse the graph simultaneously, where each processor follows a DFS-order. When all the processors are done, the subtrees grown by graph traversal are connected by the stub tree into a spanning tree. Work-stealing, a randomized work scheduling technique introduced by Blumofe and Leiserson [6], balances the graph traversals among processors and yields an expected running time that scales linearly with the number of processors for suitably large inputs. Unlike the SV approach, the labeling of vertices does not affect the performance of our new algorithm.

Generating a Stub Spanning Tree: In the first step, one processor generates a stub spanning tree, a small connected portion of the spanning tree, by randomly walking the graph for O(p) steps. The vertices of the stub spanning tree are evenly distributed into each processor's stack, and each processor in the next step traverses from the first element in its stack. After the traversals in step 2, the spanning subtrees are connected to each other by this stub spanning tree.

Performing Work-Stealing Graph Traversal: The basic idea of this step is to let each processor traverse the graph similar to the sequential algorithm in such a way that each processor finds a subgraph of the final spanning tree. In order for this step (see Alg. 2) to perform correctly and efficiently, we need to address the following two issues: 1) coloring the same vertex simultaneously by multiple processors; that is, a vertex may appear in two or more subtrees of different processors, and 2) balancing the load among the processors.

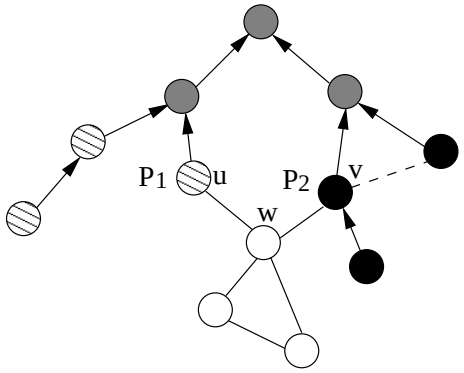

Figure 1: Two processors P_1 and P_2 work on vertex u and v, respectively. They both see vertex w as unvisited, so each is in a race to color w and set w's parent pointer. The grey vertices are in the stub spanning tree; the shaded vertices are colored by P_1; the black vertices are marked by P_2; and the white vertices are unvisited. Directed solid edges are the selected spanning tree edges; dashed edges are nontree edges; and undirected solid edges are not yet visited.

As we will show the algorithm runs correctly even when two or more processors color the same vertex. In this situation, each processor will color the vertex and set

as its parent the vertex it has just colored. Only one processor succeeds at setting the vertex's parent to a final value. For example, using Fig. 1, processor P_1 colored vertex u, and processor P_2 colored vertex v, at a certain time they both find w unvisited and are now in a race to color vertex w. It makes no difference which processor colored w last because w's parent will be set to either u or v (and it is legal to set w's parent to either of them; this will not change the validity of the spanning tree, only its shape). Further, this event does not create cycles in the spanning tree. Both P_1 and P_2 record that w is connected to each processor's own tree. When various processors visit each of w's unvisited children, its parent will be set to w, independent of w's parent.

Data : (1) An adjacency list representation of graph $G = (V, E)$ with n vertices, (2) a starting vertex *root* for each processor, (3) *color*: an array of size n with each element initialized to 0, and (4) *parent*: an array of size n.

Result: p pieces of spanning subtrees, except for the starting vertices, each vertex v has $parent[v]$ as its parent

begin

1. color my starting vertex with my label i and place it into my stack S
   $color[root] = i$
   Push(S, *root*)
2. start depth-first search from *root*, color the vertices that have not been visited with my label i until the stack is empty.

2.1 **while** Not-Empty(S) **do**
  2.2 v = Pop(S)
  2.3 **for** each neighbor w of v **do**
    2.4 **if** $(color[w] = 0)$ **then**
      2.5 $color[w] = i$
      2.6 $parent[w] = v$
      2.7 Push(S, w)

end

Algorithm 2: Graph Traversal Step for our SMP Algorithm for Processor i, $(1 \leq i \leq p)$.

Lemma 1 *On an SMP with sequential memory consistency, Alg. 2 does not create any cycles in the spanning tree.*

Proof: (by contradiction) Suppose in the SMP spanning tree algorithm processors $P_1, P_2, \cdots, P_j$ create a cycle sequence $< s_1, s_2, \cdots, s_k, s_1 >$, that is, P_i sets s_i's parent to s_{i+1}, and P_j sets s_k's parent to s_1. Here any P_i and P_j with $1 \leq i, j \leq p$ and $1 \leq k \leq n$ could be the same or different processors. According to the algorithm, s_i's parent is set to s_{i+1} only when P_i finds s_{i+1} at the top of its stack (and s_{i+1} was colored before and put into the stack), and s_i is s_{i+1}'s unvisited (uncolored) neighbor. This implies that for P_i the coloring of s_{i+1} happens before the coloring of s_i. In other words, processor P_i observes the memory write to location $color[s_{i+1}]$ happen before the write to location $color[s_i]$. On an SMP with sequential memory consistency, this means each processor should see the sequence in this order. Let t_i be the time at which s_i is colored; we have $t_i > t_{i+1}$, that is, $t_1 > t_2 > t_3 > \cdots > t_k > t_1$, which is a

contradiction. Thus, the SMP graph traversal step creates no cycles. □

Lemma 2 *For connected graph G, Alg. 2 will set parent[v] for each vertex $v \in V$ that is colored 0 before the start of the algorithm.*

Proof: First we prove (by contradiction) that each vertex with color 0 before the start of the algorithm will be colored from the set $\{1, 2, \ldots, p\}$ after the algorithm terminates. Suppose there exists a vertex $v \in V$ that still has color 0 after Alg. 2 terminates. This implies that each neighbor w of v is never placed into the stack, otherwise step 2.3 in Alg. 2 would have found that v is w's neighbor, and would have colored v as one of $1, 2, \ldots, p$. If w is never placed in the stack, then w has color 0, which in turn means that all w's neighbors have color 0. By induction, and because G is connected, we find all of the vertices in G are colored 0 after the algorithm terminates, which is clearly a contradiction. Further, since each vertex is colored, step 2.6 in Alg. 2 guarantees that each vertex's *parent* is set. □

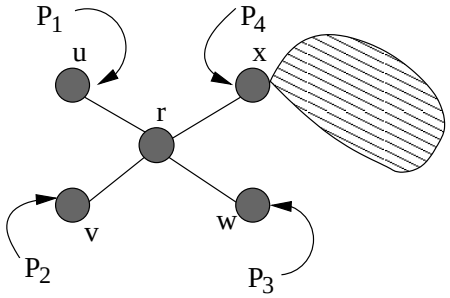

Figure 2: Unbalanced load: processors P_1, P_2, and P_3, each color only one vertex while processor P_4 colors the remaining $n - 3$ vertices.

For certain shapes of graphs or ordering of traversals, some processors may have little work to do while others are overloaded. For example, using Fig. 2, after generating a stub spanning tree (black vertices), processors P_1, P_2, P_3, and P_4, start a traversal from designated starting points. In this case P_1, P_2, and P_3, color no other vertices than u, v, and w, while processor P_4, starting from vertex x, has significant work to do. In this example for instance, this results in all but one processor sitting idle while a single processor performs almost all the work, and obviously no speedup will be achieved. We remedy this situation as follows.

To achieve better load-balancing across the processors, we add the technique of work-stealing to our algorithm. Whenever any processor finishes with its own work (that is, it cannot reach any other unvisited vertex), it randomly checks other processors' stacks. If it finds a non-empty stack, the processor steals part of the stack. Work-stealing does not affect the correctness of the algorithm, because when a processor takes elements from a stack, all of the elements are already colored and their *parent*s have already been set, and no matter which processor inspects their unvisited children, they are going to be set as these children's *parent*s. As we show later in our experimental results, we find that this technique keeps all processors equally busy performing useful work, and hence, evenly balances the workload for most classes of input graphs.

We expect that our algorithm achieves a good load-balancing on many practical applications with large diameter graphs; for instance, in infrastructure problems such as pipelines and railroads. Arguably there are still pathological cases where work-stealing could fail to balance the load among the processors. For example, when connectivity of a graph (or portions of a graph) is very low, stacks of the busy processors may only contain a few vertices. In this case work awaits busy processors while idle processors starve for something to do. Obviously this is the worst case for the SMP traversal algorithm. We argue that this case is very rare (see Section 3); however, we next propose a detection mechanism that can detect the situation and invoke a different spanning tree algorithm that is robust to this case.

The detection mechanism uses condition variables to coordinate the state of processing. Whenever a processor becomes idle and finds no work to steal, it will go to sleep for a duration on a condition variable. Once the number of sleeping processors reaches a certain threshold, we halt the SMP traversal algorithm, merge the grown spanning subtree into a supervertex, and start a different algorithm, for instance, the SV approach. In theoretic terms, the performance of our algorithm could be similar to that of SV in the worst case, but in practical terms this mechanism will almost never be triggered; for instance, in our experimental studies with a collection of different types of graphs, we never encountered such a case.

When an input graph contains vertices of degree two, these vertices along with a corresponding tree edge can be eliminated as a simple preprocessing step. Clearly, this optimization does not affect correctness of the algorithm, and we can assume that this procedure has been run before the analysis in the next section.

Theorem 1 *For connected graph G, suppose we generate a stub spanning tree and store the vertices into each processor's stack. Let each processor start the traversal from the first vertex stored in its stack. Then after the work-stealing graph traversal step terminates, we have a spanning tree of G.*

Proof: Theorem 1 follows from Lemmas 1 and 2. □

3 Analysis of the SMP Spanning Tree Algorithms

We compare our new SMP algorithm with the implementation of SV both in terms of complexity and actual performance (in Section 4). Our analyses use an SMP complexity model similar to that of Helman and JáJá [21] that has been shown to provide a good cost model for shared-memory algorithms on current symmetric multiprocessors [4, 5, 20, 21]. The model uses two parameters: the input size n, and the number p of processors. Running time $T(n,p)$ is measured by the triplet $\langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle$, where $T_M(n,p)$ is the maximum number of non-contiguous main memory accesses required by any processor, $T_C(n,p)$ is an upper bound on the maximum local computational complexity of any of the processors, and $B(n,p)$ is the number of barrier synchronizations. This model, unlike the idealistic PRAM, is more realistic in that it penalizes algorithms with non-contiguous memory accesses that often result in cache misses and algorithms with more synchronization events.

Our spanning tree algorithm takes advantage of the shared-memory environment in several ways. First, the input graph's data structure can be shared by the processors without the need for the difficult task of partitioning the input data often required by distributed-memory algorithms. Second, load balancing can be performed asynchronously using the lightweight work-stealing protocol. Like SV and HCS, the running time of our new approach is dependent on the topology of the input graph. However, while theirs are dependent on the vertex labeling, ours is not. Next, we give the complexity analyses of these approaches.

SMP Traversal Based:

The first step that generates a stub spanning tree is executed by one processor in $T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle = \langle \mathrm{O}(p)\ ;\ \mathrm{O}(p)\ ;\ 1\rangle$. In the second step, the work-stealing graph traversal step needs one non-contiguous memory access to visit each vertex, and two non-contiguous accesses per edge to find the adjacent vertices, check their colors, and set the parent. For almost all graphs, the expected number of vertices processed per processor is $\mathrm{O}\left(\frac{n}{p}\right)$ with the work-stealing technique; and hence, we expect the load to be evenly balanced. (Palmer [36] proved that almost all random graphs have diameter two.) During the tree-growing process, a small number of vertices may appear in more than one stack because of the races among the processors. Analytically, we could model this as a Poisson process that depends on parameters related to system and problem characteristics. However, this number will not be significant. Our experiments show that the number of vertices that appear in multiple processors' stacks at the same time are a miniscule percentage (for example, less than ten vertices for a graph with millions of vertices).

We expect each processor to visit $\mathrm{O}\left(\frac{n}{p}\right)$ vertices; hence, the expected complexity of the second step is $T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle = \left\langle \frac{n}{p} + 2\frac{m}{p}\ ;\ \mathrm{O}\left(\frac{n+m}{p}\right)\ ;\ 1\right\rangle$. Thus, the expected running time for our SMP spanning tree algorithm is given as

$$T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle \leq \left\langle \frac{n}{p} + 2\frac{m}{p} + \mathrm{O}(p)\ ;\ \mathrm{O}\left(\frac{n+m}{p}\right)\ ;\ 2\right\rangle, \quad (1)$$

with high probability. For realistic problem sizes ($n \gg p^2$), this simplifies to

$$T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle \leq \left\langle \mathrm{O}\left(\frac{n+m}{p}\right)\ ;\ \mathrm{O}\left(\frac{n+m}{p}\right)\ ;\ 2\right\rangle. \quad (2)$$

The algorithm scales linearly with the problem size and number of processors, and we use only a constant number of barrier synchronizations.

Shiloach-Vishkin (SV): The SV algorithm is modified from the deterministic connected components algorithms for finding spanning trees with p shared-memory processors. SV iterates from one to $\log n$ times depending on the labeling of the vertices. In the first "graft-and-shortcut" step of SV, two passes are used to ensure that a tree is not grafted onto multiple other trees. In each pass, there are two non-contiguous memory accesses per edge, for reading $D[j]$ and $D[D[i]]$. Thus, each of the two passes of the first step has cost:

$$T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle = \left\langle 2\frac{m}{p} + 1\ ;\ \mathrm{O}\left(\frac{n+m}{p}\right)\ ;\ 1\right\rangle \quad (3)$$

The second step of the SV connected components algorithm does not need to be run for spanning tree, since all the trees are grafted in the first step. The final step of each

iteration runs pointer jumping to form rooted stars ensuring that a tree is not grafted onto itself, with cost:

$$T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle = \left\langle \tfrac{n\log n}{p}\ ;\ \mathrm{O}\left(\tfrac{n\log n}{p}\right)\ ;\ 1\right\rangle \tag{4}$$

In general, SV needs multiple iterations to terminate. Assuming the worst-case of $\log n$ iterations, the total complexity for SV is

$$T(n,p) = \langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle \leq \left\langle \tfrac{n\log^2 n}{p} + \left(4\tfrac{m}{p}+2\right)\log n\ ;\ \mathrm{O}\left(\tfrac{n\log^2 n + m\log n}{p}\right)\ ;\ 4\log n\right\rangle \tag{5}$$

Comparing the analyses, we predict that the computational complexity of our randomized approach $\left(\mathrm{O}\left(\frac{n+m}{p}\right)\right)$ is asymptotically less than that of the deterministic SV approach $\left(\mathrm{O}\left(\frac{n\log^2 n+m\log n}{p}\right)\right)$. Even if SV iterates only once, there is still approximately $\log n$ times more work per iteration. Looking at memory accesses, our SMP algorithm is more cache friendly, having small number of non-contiguous memory access per the input size. On the other hand, SV has a multiplicative factor of approximately $\log^2 n$ more non-contiguous accesses per vertex assigned to each processor. Our SMP approach also uses less synchronization (O(1)) than the SV implementation that requires $\mathrm{O}(\log n)$.

4 Experimental Results

This section summarizes the experimental results of our implementation and compared our results with previous experimental results. We tested our shared-memory implementation on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 KB of direct-mapped data (L1) cache and 4 MB of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers [3].

4.1 Experimental Data

We use a collection of sparse graph generators to compare the performance of the parallel spanning tree graph algorithms. Our generators include several employed in previous experimental studies of parallel graph algorithms for related problems. For instance, we include the **2D60** and **3D40** mesh topologies used in the connected component studies of Greiner [17], Krishnamurthy *et al.* [30], Hsu *et al.* [23], and Goddard *et al.* [16], the random graphs used by Greiner [17], Chung and Condon [12], Hsu *et al.* [23], and Goddard *et al.* [16], the geometric graphs used by Chung and Condon [12], and the "tertiary" geometric graph **AD3** used by Greiner [17], Hsu *et al.* [23], Krishnamurthy *et al.* [30], and Goddard *et al.* [16]. In addition, we include generators from realistic applications such as geographic graphs and from pathological cases such as degenerate chain graphs.

- **Regular and Irregular Meshes** Computational science applications for physics-based simulations and computer vision commonly use mesh-based graphs.
  - **2D Torus** The vertices of the graph are placed on a 2D mesh, with each vertex connected to its four neighbors.

  - **2D60** 2D mesh with the probability of 60% for each edge to be present.
  - **3D40** 3D mesh with the probability of 40% for each edge to be present.

- **Random Graph** We create a random graph of n vertices and m edges by randomly adding m unique edges to the vertex set. Several software packages generate random graphs this way, including LEDA [32].
- **Geometric Graphs and AD3** In these k-regular graphs, n points are chosen uniformly and at random in a unit square in the Cartesian plane, and each vertex is connected to its k nearest neighbors. Moret and Shapiro [34] use these in their empirical study of sequential MST algorithms. **AD3** is a geometric graph with $k = 3$.
- **Geographic Graphs** Research on properties of wide-area networks model the structure of the Internet as a geographic graph [7]. We classify geographic graphs into two categories, flat and hierarchical. Flat mode takes into account the geographical locations of vertices when producing edges. First the vertices are randomly placed on a square, then for each pair of the vertices, an edge connects them according to the distance between them and other parameters. Hierarchical mode models the Internet with the notions of backbones, domains, and subdomains. Several vertices are placed in the square, and a backbone is created connecting these locations. In a similar way domains and subdomains are created around certain locations of the backbone.

4.2 Performance Results and Analysis

In this section we offer a collection of our performance results that demonstrate for the first time a parallel spanning tree algorithm that exhibits speedup when compared with the best sequential approach over a wide range of input graphs. In our SMP spanning tree algorithm, the first step generates a stub spanning tree of size $O(p)$. There is a performance trade-off between the actual size of this stub tree and the load balancing achieved by the processors. Empirically, we determined that a size of $10p$ has negligible cost to generate and was sufficiently large in our test cases to achieve a good load balance. Hence, all of our experimental results presented here generate a stub spanning tree of size $10p$. We expect the performance of our algorithm to vary due to the randomization in the stub spanning tree generation and work-stealing load balancing. Our empirical results showed less than 5% variations between runs on the same graph instance for those graphs we tested. Because this variance is minimal, we plot our experimental results as an average over 10 runs on the same instance.

The performance plots in Fig. 3 are for the regular and irregular meshes (torus, **2D60** and **3D40**), in Fig. 4 are for the random, geometric and **AD3**, and geographic classes of graphs, and in Fig. 5 are for the degenerate chain graphs. Note that only the mesh and degenerate chain graphs are regular; all of the remaining graphs used are irregular. In these plots, the horizontal dashed line represents the time taken for the best sequential spanning tree algorithm to find a solution on the same input graph using a single processor of the Sun E4500. Throughout this paper, our sequential algorithm use an optimized implementation of depth-first search to find a spanning tree.

In the case of the torus inputs, we observe that the initial labeling of vertices greatly affects the performance of the SV algorithm, but the labeling has little impact on our

algorithm. In all of these graphs, we note that the SV approach runs faster as we employ more processors. However, in many cases, the SV parallel approach is slower than the best sequential algorithm. For $p > 2$ processors, in our testing with a variety of classes of large graphs, our new spanning tree algorithm is always faster than the sequential algorithm, and executes faster as more processors are available. This is remarkable, given that the sequential algorithm runs in linear time with a very small hidden constant in the asymptotic complexity.

Fig. 6 shows the scalability of our SMP spanning tree algorithm and SV using $p = 8$ processors compared with the sequential algorithm for a random graph with $m = 4n$ edges. For these experiments, the speedup of our algorithm compared with the sequential approach ranges between 3.8 and 4.1, while SV is slower than the sequential approach.

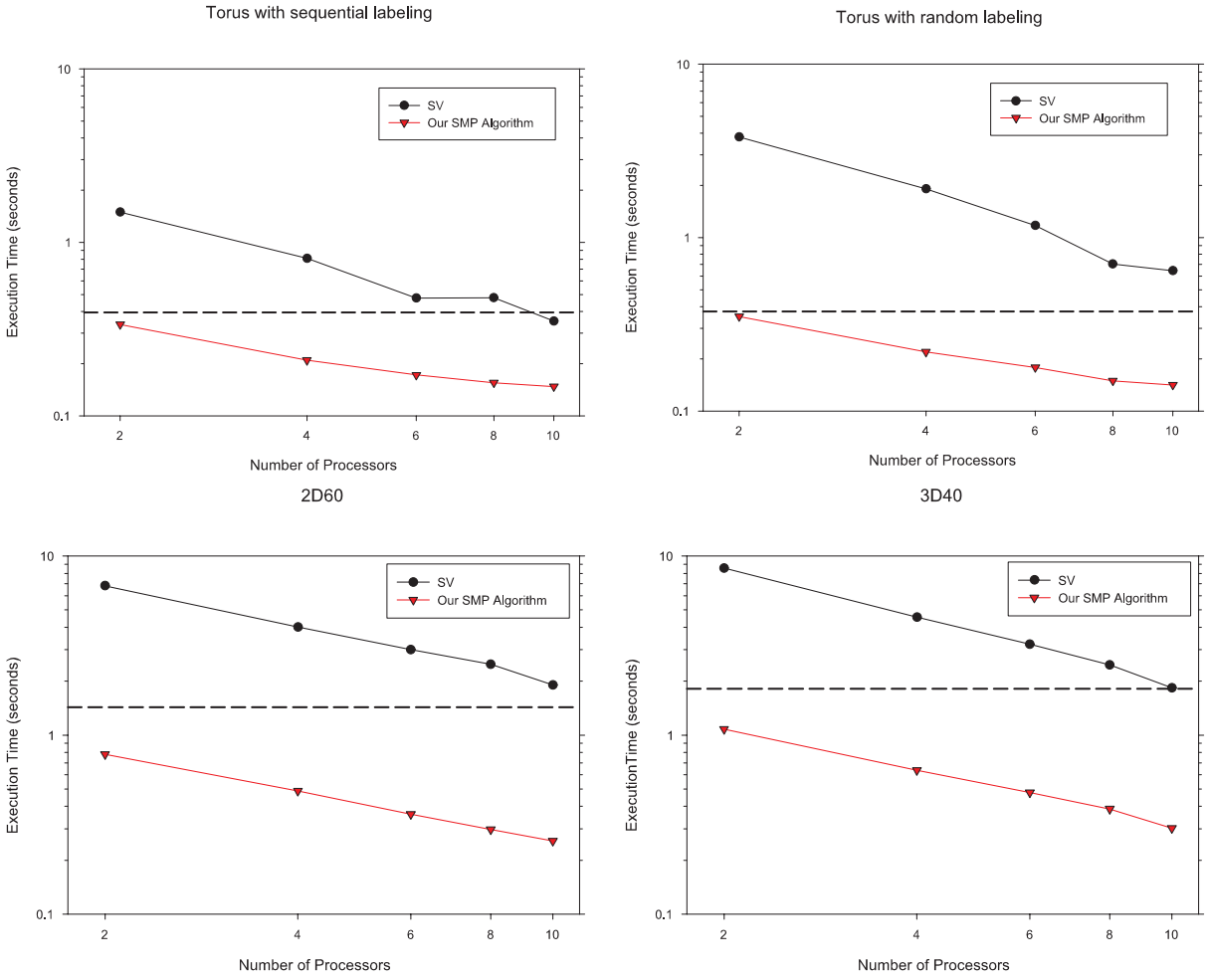

Figure 3: Comparison of parallel spanning tree algorithms for regular and irregular meshes with $n = 2^{20}$ vertices. The top-left plot uses a row-major order labeling of the vertices in the torus, while the top-right plot uses a random labeling. The bottom-left and -right plots are for irregular torus graphs **2D60** and **3D40**, respectively. The dashed line corresponds to the best sequential time for solving the input instance. Note that these performance charts are log-log plots.

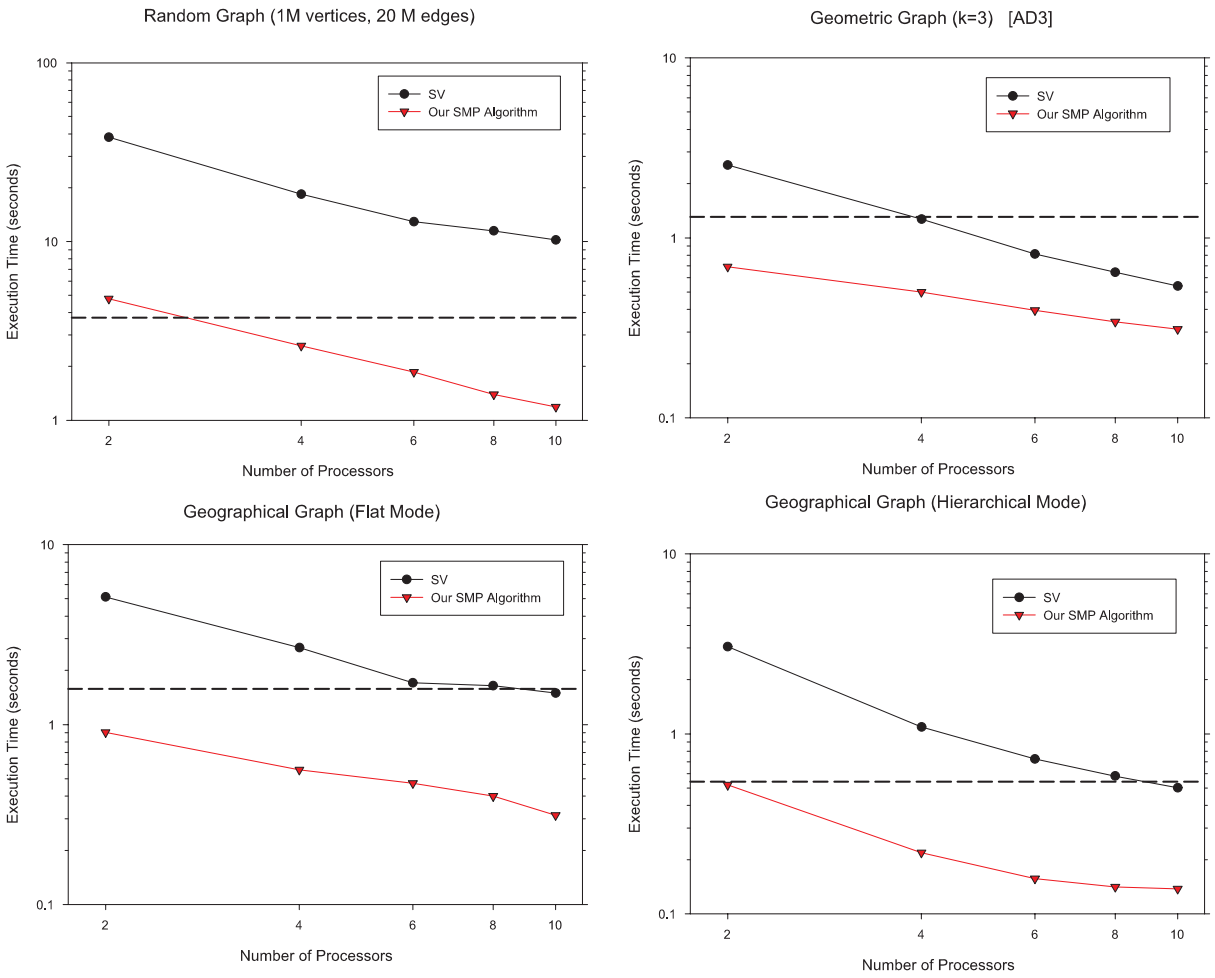

Figure 4: Comparison of parallel spanning tree algorithms for graphs with $n = 2^{20}$ vertices. The top-left plot uses a random graph with $m = 20M \approx n \log n$ edges. The top-right plot uses **AD3**, a geometric graph with $k = 3$. The bottom-left and -right plots are for geographic inputs with flat and hierarchical modes, respectively. The dashed line corresponds to the best sequential time for solving the input instance. Note that these performance charts are log-log plots.

5 Conclusions and Future Work

In summary, we present optimistic results that for the first time, show that parallel spanning tree algorithms run efficiently on parallel computers for graphs with regular and irregular topologies. Our new implementation scales nearly linearly with the problem size and the number of processors for suitably large input graphs. Our randomized approach uses a load balancing scheme based upon work-stealing and exhibits superior performance when compared with prior deterministic parallel approaches that we modify for SMPs. Through comparison with the best sequential implementation, we see experimentally that our approach runs in $O\left(\frac{n+m}{p}\right)$ expected time over a variety of reg-

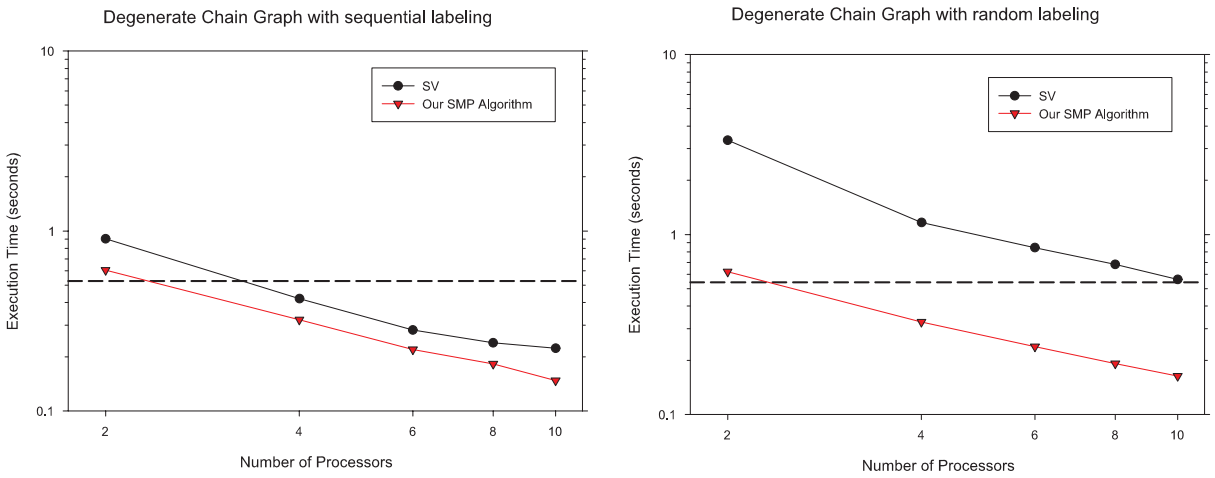

Figure 5: Comparison of parallel spanning tree algorithms for graphs with $n = 2^{20}$ vertices. The left plot uses a degenerate graph with a sequential labeling of the vertices, while the right plot uses a random labeling. Note that the performance of our parallel spanning tree algorithm is unaffected by the labeling. The dashed line corresponds to the best sequential time for solving the input instance. Note that these performance charts are log-log plots.

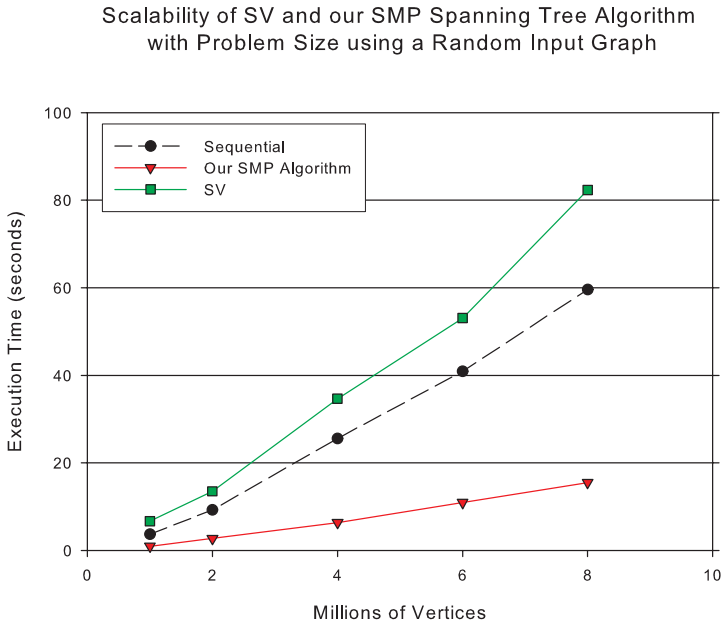

Figure 6: This plot shows the scalability of SV and our SMP spanning tree algorithm using $p = 8$ processors compared with the sequential algorithm for a random graph. For these experiments, the speedup of our SMP spanning tree algorithm compared with the sequential approach is between 3.8 and 4.1, while SV never beats the sequential implementation.

ular and irregular graph topologies. Further, these results provide optimistic evidence that complex graph problems that have efficient PRAM solutions, but often no known efficient parallel implementations, may scale gracefully on SMPs. Our future work includes validating these experiments on larger SMPs, and since the code is portable, on other commercially-available platforms. We plan to apply the techniques discussed in this paper to other related graph problems, for instance, minimum spanning tree (forest), connected components, and planarity testing algorithms.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258–1263, 1987.

[2] 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.

[3] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[4] 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.

[5] 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.

[6] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[7] K.L. Calvert, M.B. Doar, and E.W. Zegura. Modeling internet topology. *IEEE Communications*, 35(6):160–163, 1997.

[8] A. Charlesworth. Starfire: extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998.

[9] A. Charlesworth. The Sun Fireplane system interconnect. In *Proc. Supercomputing (SC 2001)*, pages 1–14, Denver, CO, November 2001.

[10] F. Y. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[11] K.W. Chong and T.W. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *J. Algorithms*, 18:378–402, 1995.

[12] 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.

[13] R. Cole and U. Vishkin. Approximate parallel scheduling. part II: applications to logarithmic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[14] F. Dehne and S. Götz. Practical parallel algorithms for minimum spanning trees. In *Workshop on Advances in Parallel and Distributed Systems*, pages 366–371, West Lafayette, IN, October 1998.

[15] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.

[16] 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.

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

[19] Y. Han and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990.

[20] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[21] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[22] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[23] 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.

[24] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *J. Algorithms*, 16(2):190–217, 1994.

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

[26] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} |v|)$ parallel time for the CREW PRAM. In *Proc. of the 32nd Ann. IEEE Symp. on Foundations of Computer Science*, pages 688–697, San Juan, Puerto Rico, 1991.

[27] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proc. 4th Ann. Symp. Parallel Algorithms and Architectures (SPAA-92)*, pages 363–372, San Diego, CA, 1992.

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

[29] P.N. Klein and J.H. Reif. An efficient parallel algorithm for planarity. *J. Computer and System Sciences*, 37(2):190–246, 1988.

[30] 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.

[31] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5(1):43–64, 1990.

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

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

[34] 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.

[35] D. Nash and S.N. Maheshwari. Parallel algorithms for the connected components and minimal spanning trees. *Information Processing Letters*, 14(1):7–11, 1982.

[36] E. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs*. John Wiley & Sons, New York, 1985.

[37] C.A. Phillips. Parallel graph contraction. In *Proc. 1st Ann. Symp. Parallel Algorithms and Architectures (SPAA-89)*, pages 148–157, Santa Fe, NM, 1989. ACM.

[38] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[39] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[40] J.H. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR-08-85, Harvard Univ., Boston, MA, March 1985.

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