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

David A. Bader* Guojing Cong

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

Abstract

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 often have no known efficient parallel implementations. 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, and ITR EF/BIO 03-31654.

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, and can be used in graph planarity testing. 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, whose time complexity is $O(m+n)$. The implementation of the sequential algorithm is 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 is inherently sequential and not known 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 irregular graphs.

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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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) [7, 8], 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.

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 algorithms include the following results. **EREW PRAM:**: Nash and Maheshwari's approach [33] takes $O(\log^2 n)$ time with $O(n^2)$ operations and improves from Hirschberg, Chandra and Sarwate's CREW algorithm [22] by building data structures to eliminate the current reads; Phillips [36] and Kruskal, Rudolph, and Snir [30] gave algorithms that run in $O(\log^2 n)$ time with $O((m+n)\log n)$ work; Chong and Lam's approach [10] takes $O(\log n \log\log n)$ time with $O((m+n)\log n \log\log n)$ work; and is further improved by Chong, Han, and Lam [11] with a $O(\log n)$ time and $O(n\log n)$ work algorithm. **CREW PRAM:** Hirschberg *et al.* [22] designed an $O(\log^2 n)$ time, $O(n^2 \log n)$ work algorithm; Chin, Lam, and Chen [9] improved the algorithm to take $O(\log^2 n)$ time and $O(n^2)$ work by exploiting the adjacency matrix as the representing data structure; Han and Wagner's algorithm [19] runs in $O(\log^2 n)$ time with $O((m+n\log n)\log n)$ work; and Johnson and Metaxas's algorithm [26, 27] runs in $O\left(\log^{3/2} n\right)$ time with $O\left((m+n)\log^{3/2} n\right)$ operations. **CRCW PRAM:** There are fast algorithms that use concurrent writes. Shiloach and Vishkin [40] and Awerbuch and Shiloach [1] developed algorithms that run in $O(\log n)$ time with $O((m+n)\log n)$ work; both Cole and Vishkin's algorithm [13] and Iwama and Kambayashi's algorithm [24] run in $O(\log n)$ time with $O((m+n)\alpha(m,n))$ work, where α is the inverse Ackermann's function. 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.

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] is actually a mixture of several previous algorithms; it borrows the *maximum-hooking* method from Chong and Lam [10] 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]. In [35] Pettie and Ramachandran give an optimal randomized minimum spanning tree algorithm that is simpler than that of Halperin and Zwick.

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, to our knowledge there is no parallel implementation of spanning tree (or the related problems such as 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.* [29], and Hsu *et al.* [23]), and compared these with our new randomized approach. Our results to our knowledge are 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 [37] and Phillips [36], 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, and about half as fast as Greiner's CM-2 code, but Hsu *et al.*'s implementation uses one-fourth of the total memory used by Greiner's hybrid approach. Krishnamurthy *et al.* [29] implemented a connected components algorithm (based on Shiloach-Vishkin [40]) 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]

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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 algorithms 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.

2. Parallel Spanning Tree Algorithms for SMPs

Here we present an overview of the three parallel spanning tree algorithms we have implemented. We refer the reader to [4] for further detail and proof of correctness. According to the complexities of the algorithms, programming complexity, constant factors hidden in the asymptotic notation, we choose two representative PRAM algorithms to implement for SMPs, i.e, the Shiloach-Vishkin (SV) and the Hirschberg-Chandra-Sarwate (HCS) algorithms, using appropriate optimizations suggested by [17, 23, 29, 12]. 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. Through the experience we gained by implementing these two algorithms, we developed a new randomized algorithm with superior performance in all of our experiments.

The Shiloach-Vishkin Algorithm:: The Shiloach-Vishkin algorithm (SV) is in fact a connected-components algorithm [40, 1]. 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 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. On the priority CRCW model (assumed by the original algorithm) arbitration among the processors during each step is provided by the model, yet most SMPs can only provide arbitrary concurrent writes with a cost. 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 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 performing 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 $O(\log^2 n)$; the additional $\log n$ factor comes from shortcutting (pointer jumping). Optimizations are possible for the election approach. Please see [4] for additional optimization details.

SV is sensitive to the labeling of vertices, since alternative labelings of the 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, the number of iterations needed will be from one to $\log n$. We expect to see similar behaviors for the class of algorithms that use the "grafting and short-cutting" approach.

A New Spanning Tree Algorithm For SMPs: Our new parallel spanning tree algorithm for shared-memory multiprocessors has two main steps: 1) stub spanning tree, and 2) work-stealing graph traversal. Work-stealing is a randomized technique used for load balancing the graph traversals 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.

Stub Spanning Tree: In the first step, one processor generates a stub spanning tree, that is, a small 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 queue, and each processor traverses from the first element in its queue.

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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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 queue Q_i
   color $[root] = i$
   Enqueue(Q_i,*root*)
2. start breadth-first search from *root*, color the vertices that have not been visited with my label i until queue Q_i is empty.

2.1 **while** *Not-Empty*(Q_i) **do**
  2.2 $v =$ Dequeue(Q_i)
  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 Enqueue(Q_i,w)

end

Algorithm 1: Modified Graph Traversal Step for our SMP Algorithm for Processor i, for $(1 \leq i \leq p)$. Note that the starting vertex *root* for each processor that initially is held in each processor's queue Q_i is the corresponding vertex from the stub spanning tree.

a subgraph of the final spanning tree. (After the traversals, the spanning subtrees are connected by the stub spanning tree.) In order for this step (see Alg. 1) 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.

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

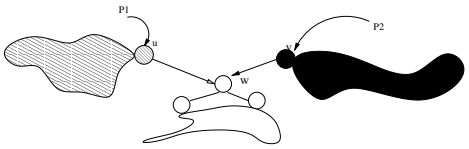

Figure 1. Two processors P_1 and P_2 see vertex w as unvisited, so each is in a race to color w and set w's parent pointer. The shaded area represents vertices colored by P_1, the black area represents those marked by P_2, and the white area contains unvisited vertices.

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.

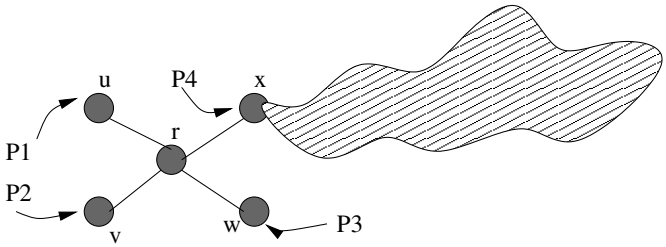

Figure 2. Unbalanced load: processors P_1, P_2, and P_3, each color only one vertex while processor P_4 colors the remaining $n-3$ vertices.

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, i.e., whenever any processor finishes with its own work (that is, it cannot reach any other unvisited vertex), it randomly checks other processors' queues. If it finds a non-empty queue, the processor steals part of the queue. Work stealing does not affect the correctness of the algorithm, because when a processors takes elements from a queue, all of the elements are already colored and their *parent*s have already been set, and no matter which processor inspects their un-

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

visited 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.

Arguably there are still pathological cases where load-balancing could fail to balance the load among the processors. For example, when connectivity of a graph (or portions of a graph) is very low (the diameter of the graph is large), queues of the busy processors may contain only a few elements (in extreme cases, the queue of a busy processor could contain only one element). In this case work awaits busy processors while idle processors starve. 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 detecting 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 super-vertex, 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.

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 [20, 21, 2, 3]. 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, in comparison with 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 data partitioning often required by distributed-memory algorithms. Second, load balancing can be performed asynchronously using the lightweight work stealing protocol. Unlike the SV implementation that is sensitive to both the labeling and topology of input graph, the running time of our new approach is dependent on the topology and nondeterministic races amongst processors. 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 [34] 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 queue 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' queues 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 $T(n,p) \leq \left\langle 5\frac{n}{p} + 2\frac{m}{p} + \mathrm{O}(p)\,;\,\mathrm{O}\left(\frac{n+m}{p}\right)\,;\,2\right\rangle$, with high probability. For realistic problem sizes ($n \gg p^2$), this simplifies to $T(n,p) \leq \left\langle \mathrm{O}\left(\frac{n+m}{p}\right)\,;\,\mathrm{O}\left(\frac{n+m}{p}\right)\,;\,2\right\rangle$, the algorithm scales linearly with the problem size and number of processors, and we use only a constant number of barrier synchronizations.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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. Thus, each of the two passes of the first step has cost $T(n,p) = \left\langle 2\frac{m}{p}+1 \; ; \; O\left(\frac{n+m}{p}\right) \; ; \; 1 \right\rangle$. 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 to ensure that a tree is not grafted onto itself, with cost $T(n,p) = \left\langle \frac{n\log n}{p} \; ; \; O\left(\frac{n\log n}{p}\right) \; ; \; 1 \right\rangle$. In general, SV needs multiple iterations to terminate. Assuming the worst-case of $\log n$ iterations, the total complexity for SV is $T(n,p) \leq \left\langle \frac{n\log^2 n}{p} + \left(4\frac{m}{p}+2\right)\log n \; ; \; O\left(\frac{n\log^2 n + m\log n}{p}\right) \; ; \; 4\log n \right\rangle$.

Comparing the analyses, we predict that our randomized approach has less computation $\left(O\left(\frac{n+m}{p}\right)\right)$ than the deterministic SV approach that has worst-case computational requirements of $O\left(\frac{n\log^2 n + m\log n}{p}\right)$. Even if SV iterates only once, there is still approximately $\log n$ times more work per iteration. Considering memory accesses, our SMP algorithm is more cache friendly, having a small number of non-contiguous memory accesses per the input size. On the other hand, SV has a multiplicative factor of approximately $\frac{1}{5}\log^2 n$ 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 $O(\log n)$.

4. Experimental Results

This section summarizes the experimental results of our implementation and compares 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 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers [5].

Experimental Data: Next we describe the collection of sparse graph generators that we use 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.* [29], 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], and 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.* [29], 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 [31].
- **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 [32] 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 [6]. 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.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

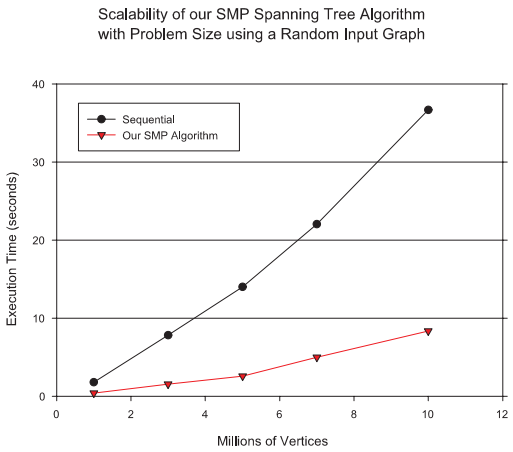

Figure 3. This plot shows the scalability of our SMP spanning tree algorithm using $p = 8$ processors compared with the sequential algorithm for a random graph. For these experiments, the speedup of the parallel algorithm is between 4.5 and 5.5.

Performance Results and Analysis: The performance plots in Fig. 4 are for the regular and irregular meshes (torus, **2D60** and **3D40**), the random, geometric and **AD3**, and geographic classes of graphs, and 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 line labeled "Sequential" represents the time taken for the best sequential spanning tree algorithm—in our experiments, breadth-first search—to find a solution on the same input graph using a single processor on the Sun E4500.

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 is linear time with a very small hidden constant in the asymptotic complexity.

Fig. 3 shows the scalability of our SMP spanning tree algorithm using $p = 8$ processors compared with the sequential algorithm for a random graph with $m = 1.5n$ edges. For these experiments, the speedup of the parallel algorithm compared with the sequential approach ranges between 4.5 and 5.5.

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 symmetric multiprocessors 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. Our new parallel algorithm has 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 regular 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 vendors' 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, for symmetric multiprocessors.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

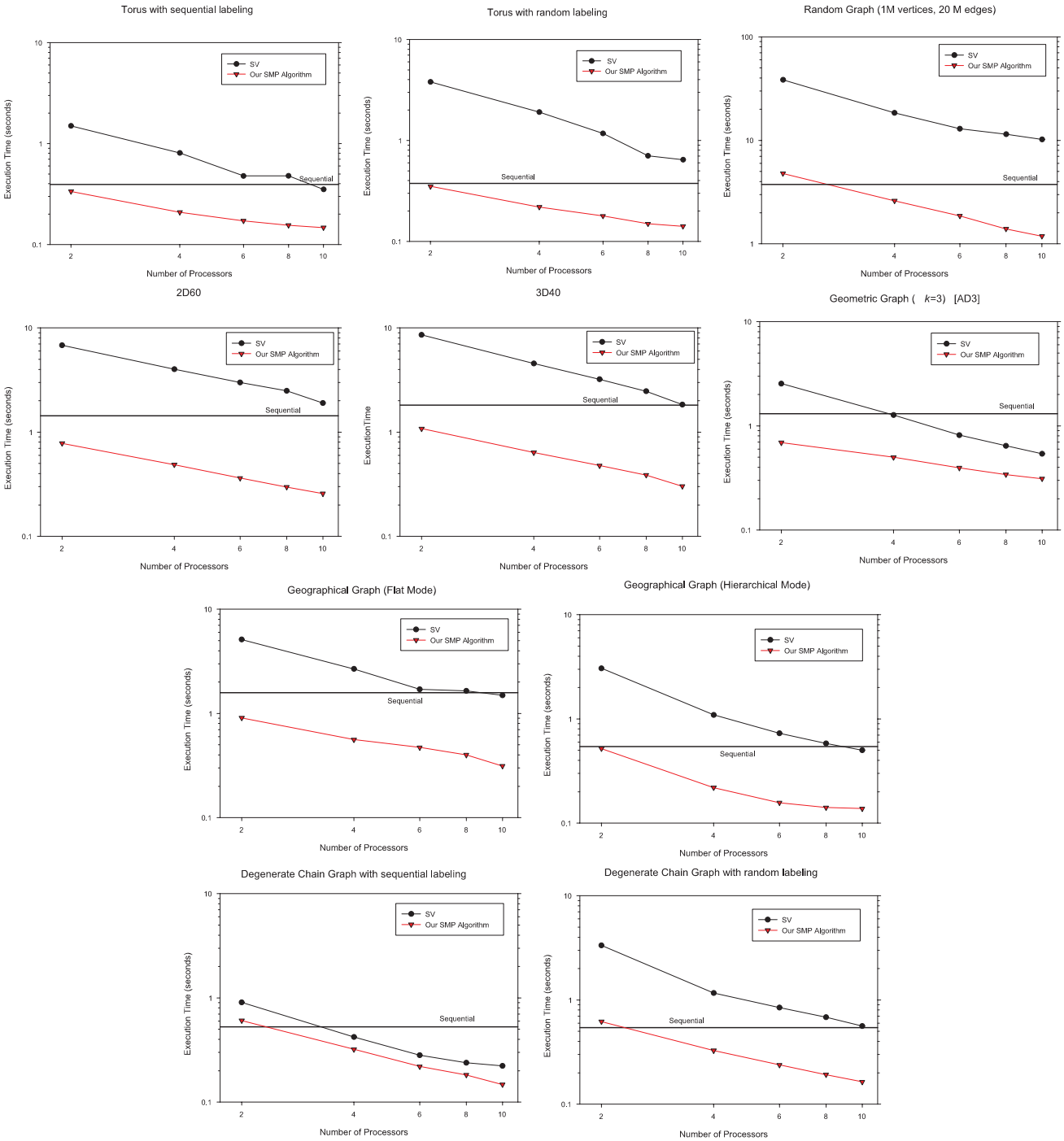

Figure 4. Comparison of parallel spanning tree algorithms for regular and irregular graphs with $n = 1M$ vertices. The top row of plots uses tori with row-major order and random labeling of the vertices followed by a random graph with $m = 20M \approx n \log n$ edges. The second row uses two irregular torus graphs 2D60 and 3D40, and a geometric graph with $k = 3$ (AD3). The third row plots are for geographic inputs with flat and hierarchical modes. The bottom row plots use a degenerate graph with a sequential and random labeling of the vertices. Note that these performance charts are log-log plots.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

References

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

[2] D. Bader, A. Illendula, B. M. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G. 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.

[3] D. 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. 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, Dec. 2002. Springer-Verlag.

[4] D. A. Bader and G. Cong. A new, fast parallel spanning tree algorithm for symmetric multiprocessors (SMPs). Technical Report 03-001, Electrical and Computer Engineering Department, University of New Mexico, Mar. 2003.

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

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

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

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

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

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

[11] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning tree algorithm. *J. ACM*, 48:297–323, 2001.

[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, Apr. 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, Oct. 1998. co-located with the 17th IEEE Symp. on Reliable Distributed Systems.

[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. 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. *J. 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, Jan. 1999. Springer-Verlag.

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

[22] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. 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. Johnson and P. Metaxas. Connected components in $O(\log^{3/2}|v|)$ parallel time for the CREW PRAM. In *Proc. of the 32nd Annual IEEE Symp. on Foundations of Computer Science*, pages 688–697, San Juan, Puerto Rico, 1991.

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

[29] 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.

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

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

[32] B. Moret and H. 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.

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

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

[35] 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.

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

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

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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE