Fast Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs

David A. Bader[*]   Guojing Cong

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

October 3, 2003

Abstract

Minimum Spanning Tree (MST) is one of the most studied combinatorial problems with practical applications in VLSI layout, wireless communication, and distributed networks, recent problems in biology and medicine such as cancer detection, medical imaging, and proteomics, and national security and bioterrorism such as detecting the spread of toxins through populations in the case of biological/chemical warfare. Most of the previous attempts for improving the speed of MST using parallel computing are too complicated to implement or perform well only on special graphs with regular structure. In this paper we design and implement four parallel MST algorithms (three variations of Borůvka plus our new approach) for arbitrary sparse graphs that for the first time give speedup when compared with the *best* sequential algorithm. In fact, our algorithms also solve the minimum spanning forest problem. We provide an experimental study of our algorithms on symmetric multiprocessors such as IBM's p690/Regatta and Sun's Enterprise servers. Our new implementation achieves good speedups over a wide range of input graphs with regular and irregular structures, including the graphs used by previous parallel MST studies. For example, on an arbitrary random graph with 1M vertices and 20M edges, we have a speedup of 5 using 8 processors. The source code for these algorithms is freely-available from our web site `hpc.ece.unm.edu`.

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

[*] 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

Given an undirected connected graph G with n vertices and m edges, the minimum spanning tree (MST) problem finds a spanning tree with the minimum sum of edge weights. MST is one of the most studied combinatorial problems with practical applications in VLSI layout, wireless communication, and distributed networks [23, 31, 32], recent problems in biology and medicine such as cancer detection [4, 19, 20, 22], medical imaging [2], and proteomics [26, 11], and national security and bioterrorism such as detecting the spread of toxins through populations in the case of biological/chemical warfare [5].

While several theoretic results are known for solving MST in parallel, many are considered impractical because they are too complicated and have large constant factors hidden in the asymptotic complexity. Pettie and Ramachandran [28] designed a randomized, time-work optimal MST algorithm for the EREW PRAM. Cole *et. al.* [9, 8] and Poon and Ramachandran [29] earlier had randomized linear-work algorithms on CRCW PRAM and EREW PRAM. Chong, Han and Lam [6] gave a deterministic EREW PRAM algorithm that runs in logarithmic time with a linear number of processors. On the BSP model, Adler *et. al.* [1] presented a communication-optimal MST algorithm. Katriel *et. al.* [18] have recently developed a new pipelined algorithm that uses the cycle property and provide an experimental evaluation on the special-purpose NEC SX-5 vector computer. In this paper we present our implementations of MST algorithms on shared-memory multiprocessors that achieve for the first time in practice reasonable speedups over a wide range of input graphs, including arbitrary sparse graphs, a challenging problem. In fact, if G is not connected, our algorithms find the MST of each connected component, hence solving the minimum spanning forest problem.

We start with the design and implementation of a parallel Borůvka's algorithm. Borůvka's algorithm is one of the earliest MST approaches, and the Borůvka iteration (or its variants) serves as a basis for several of the more complicated parallel MST algorithms, hence its efficient implementation is critical for parallel MST. Three steps characterize a Borůvka iteration: *find-min*, *connect-components*, and *compact-graph*. *Find-min* and *connect-components* are simple and straightforward to implement, and the *compact-graph* step performs bookkeeping that is often left as a trivial exercise to the reader. JáJá [16] describes a compact-graph algorithm for dense inputs. For sparse graphs, though, the compact-graph step often is the most expensive step in the Borůvka iteration. Section 2 explores different ways to implement the compact-graph step, then proposes a new data structure for representing sparse graphs that can dramatically reduce the running time of the compact-graph step with a small cost to the find-min step. The analysis of these approaches is given in Section 3.

In Section 4 we present a new parallel MST algorithm for symmetric multiprocessors (SMPs) that marries the Prim and Borůvka approaches. In fact, the algorithm when run on one processor behaves as Prim's, and on n processors becomes Borůvka's, and runs as a hybrid combination for $1 < p < n$, where p is the number of processors.

Our target architecture is symmetric multiprocessors (SMPs). Most of the new high-performance computers are clusters of SMPs having from 2 to over 100 processors per node. In SMPs, processors operate in a true, hardware-based, shared-memory environment. SMP computers bring us much closer to PRAM, yet it is by no means the PRAM used in theoretical work—synchronization cannot be taken for granted, memory bandwidth is limited, and the number of processors is far smaller than that assumed in PRAM algorithms. Designing and implementing parallel algorithms for SMPs requires special considerations that are crucial to a fast and efficient implementation. For example, memory bandwidth often limits the scalability and locality must be exploited to make good use of cache. This paper presents the first results of actual parallel speedup for finding an MST of irregular, arbitrary sparse graphs when compared to the best known sequential algorithm. In Section 5 we detail the experimental evaluation, describe the input data sets and testing environment, and present the empirical results. Finally, Section 6 provides our conclusions and future work.

1.1 Related Experimental Studies

Although several fast PRAM MST algorithms exist, to our knowledge there is no parallel implementation of MST that achieves significant speedup on sparse, irregular graphs when compared against the best sequential implementation.

Chung and Condon [7] implement parallel Borůvka's algorithm on the CM-5. On a 16-processor machine, for geometric, structured graphs with 32,000 vertices and average degree 9 and graphs with fewer vertices but higher average degree, their code achieve a relative parallel speedup of about 4, on 16-processors, over the sequential Borůvka's algorithm, which was already 2–3 times slower than their sequential Kruskal algorithm.

Dehne and Götz [10] studied practical parallel algorithms for MST using the BSP model. They implement 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 achieve a maximum speedup of 6.1 using 16 processors for a random dense graph. Their algorithm is not suitable for the more challenging sparse graphs.

2 Designing Data Structures for Parallel Borůvka's Algorithms on SMPs

Borůvka's minimum spanning tree algorithm lends itself more naturally to parallelization, since other approaches like Prim and Kruskal are inherently sequential, with Prim growing a single MST one branch at a time, while Kruskal scanning the graph's edges in a linear fashion. Three steps comprise each iteration of parallel Borůvka's algorithm:

1. **find-min**: for each vertex v label the incident edge with the smallest weight to be in the MST.
2. **connect-components**: identify connected components of the induced graph with edges found in Step 1.
3. **compact-graph**: compact each connected component into a single supervertex, remove self-loops and multiple edges; and re-label the vertices for consistency.

Steps 1 and 2 (find-min and connect-components) are relatively simple and straightforward; in [7], Chung and Condon discuss an efficient approach using pointer-jumping on distributed memory machines, and load balancing among the processors as the algorithm progresses. Simple schemes for load-balancing suffice to distribute the work roughly evenly to each processor. For pointer-jumping, although the approaches proposed in [7] can be applied to shared-memory machines, experimental results show that this step only takes a small fraction of the total running time.

Step 3 (compact-graph) shrinks the connected components and relabels the vertices. For dense graphs that can be represented by an adjacency matrix, JáJá [16] describes a simple and efficient implementation for this step. For sparse graphs this step often consumes the most time yet no detailed discussion appears in the literature. In the following subsections we describe our design of three Borůvka approaches that use different data structures, and compare the performance of each implementation.

2.1 Bor-EL: Edge List Representation with Global Edge Sort

In this implementation of Borůvka's algorithm (designated **Bor-EL**), we use the edge list representation of graphs, with each edge (u,v) appearing twice in the list for both directions (u,v) and (v,u). An elegant implementation of the compact-graph step sorts the edge list (using an efficient parallel sample sort [14]) with the supervertex of the first endpoint as the primary key, the supervertex of the second endpoint as the secondary key, and the edge weight as the tertiary key. When sorting completes, all of the self-loops and multiple edges between two supervertices appear in consecutive locations, and can be merged efficiently using parallel prefix-sums.

2.2 Bor-AL: Adjacency List Representation with Two-Level Sort

With the adjacency list representation (but using the more cache-friendly adjacency arrays [27]) each entry of an index array of vertices points to a list of its incident edges. The compact-graph step first sorts the vertex array according to the supervertex label, then concurrently sorts each vertex's adjacency list using the supervertex of the other endpoint of the edge as the key. After sorting, the set of vertices with the same supervertex label are contiguous in the array, and can be merged efficiently. We call this approach **Bor-AL** .

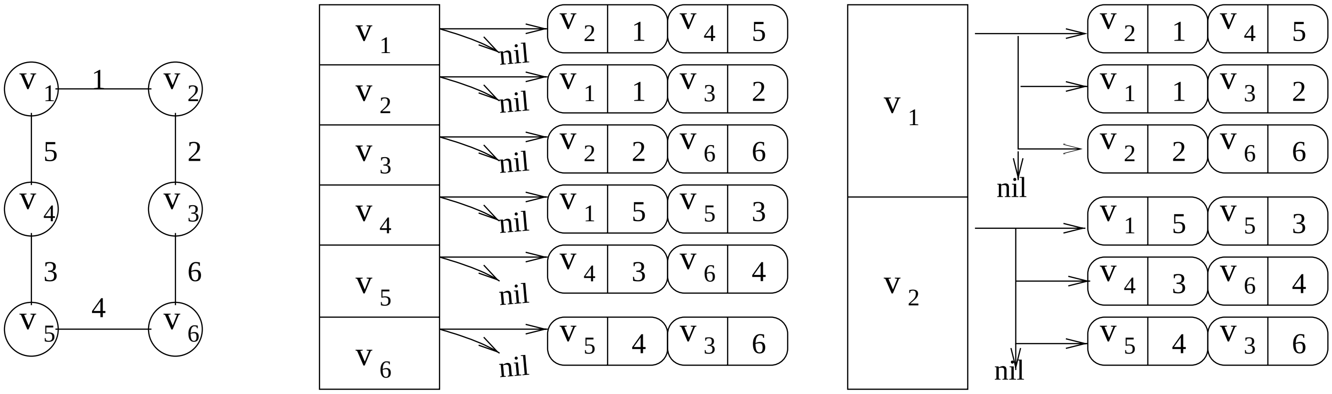
Both **Bor-EL** and **Bor-AL** achieve the same goal that self-loops and multiple edges are moved to consecutive locations to be merged. **Bor-EL** uses one call to sample sort while **Bor-AL** calls a smaller parallel sort and then a number of concurrent sequential sorts. We make the following algorithm engineering choices for the sequential sorts used in this approach. The $\mathrm{O}(n^2)$ insertion sort is generally considered a bad choice for sequential sort, yet for small inputs, it outperforms $\mathrm{O}(n \log n)$ sorts. Profiling shows that there could be many short lists to be sorted for very sparse graph. For example, for one of our input random graph with 1M vertices, 6M edges, 80% of all 311,535 lists to be sorted have between 1 to 100 elements. We use insertion sort for these short lists. For longer lists we use a non-recursive $\mathrm{O}(n \log n)$ merge sort.

Bor-ALM is an alternative adjacency list implementation of Borůvka's approach for Sun Solaris 9 that uses our own memory management routines for dynamic memory allocation rather than using the system heap. While the algorithm and data structures in **Bor-ALM** are identical to that of **Bor-AL** , we allocate private data structures using a separate memory segment for each thread to reduce contention to kernel data structures, rather than using the system `malloc()` that manages the heap in a single segment and causes contention for a shared kernel lock.

2.3 Bor-FAL: Flexible Adjacency List Representation

For the previous two approaches, conceivably the compact-graph could be the most expensive step for a parallel Borůvka's algorithm. Next we propose an alternative approach with a new graph representation data structure (that we call *flexible adjacency list*) that significantly reduces the cost for compacting the graph.

The flexible adjacency list augments the traditional adjacency list representation by allowing each vertex to hold multiple adjacency lists instead of just a single one; in fact it is a linked list of adjacency lists (and similar to **Bor-AL** , we use the more cache-friendly adjacency array for each list). During initialization, each vertex points to only one adjacency list. After the connect-components step, each vertex appends its adjacency list to its supervertex's adjacency list by sorting together the vertices that are labeled with the same supervertex. We simplify the compact-graph step, allowing each supervertex to have self-loops and multiple edges inside its adjacency list. Thus, the compact-graph step now uses a smaller parallel sort plus several pointer operations instead of costly sortings and memory copies, while the find-min step gets the added responsibility of filtering out the self-loops and multiple edges. Note that for this new approach (designated **Bor-FAL**) there are potentially fewer memory write operations compared with the previous two approaches. This is important for an implementation on SMPs because memory writes typically generate more cache coherency transactions than do reads.



a. Input Graph b. Initialized flexible adjacency list c. Flexible adjacency list after one iteratior

Figure 1: Example of Flexible Adjacency List Representation

In Fig. 1 we illustrate the use of flexible adjacency list for a 6-vertex input graph. After one Borůvka iteration, vertices 1, 2, and 3, form one supervertex and vertices 4, 5, and 6, form a second supervertex. Vertex labels 1 and 4 represent the supervertices and receive the adjacency lists of vertices 2 and 3, and vertices 5 and 6, respectively. Vertices 1 and 4 are re-labeled as 1 and 2. Note that most of the original data structure is kept intact so that we might save memory copies.

Instead of re-labeling vertices in the adjacency list, we maintain a separate lookup table that holds the supervertex label for each vertex. We easily obtain this table from the connect-components step. The find-min step uses this table to filter out self-loops and multiple edges.

3 Analysis

Here we analyze the complexities of the different Borůvka variants. Helman and JáJá's SMP complexity model [14] provides a reasonable framework for the realistic analysis that favors cache-friendly algorithms by penalizing non-contiguous memory accesses. Under this model, there are two parts to an algorithm's complexity, M_E the memory access complexity and T_C the computation complexity. The M_E term is the number of non-contiguous memory accesses, the T_C term is the running time. The M_E term recognizes the effect that memory accesses have over an algorithm's performance. Parameters of the model includes the problem size n and the number of processors p.

For a sparse graph G with n vertices and m edges, as the algorithm iterates, the number of vertices decreases by at least half in each iteration, so there are in total $\log n$ iterations for all of the Borůvka variants.

First we consider the complexity of **Bor-EL** . The find-min and connect-component steps are straightforward, their aggregate complexity in one iteration is (assuming balanced load among processors) is characterized by:

$$T(n,p) = \langle M_E \; ; \; T_C \rangle = \left\langle \frac{n + n\log n}{p} \; ; \; \mathrm{O}\left(\tfrac{m+n\log n}{p}\right) \right\rangle . \tag{1}$$

The parallel sample sort that we use in **Bor-EL** for compact-graph has the complexity of

$$T(n,p) = \langle M_E \; ; \; T_C \rangle = \left\langle \left(4 + 2\frac{c\log\frac{l}{p}}{\log z}\right)\frac{l}{p} \; ; \; \mathrm{O}\left(\tfrac{l}{p}\log l\right) \right\rangle \tag{2}$$

with high probability where l is the length of the list and c and z are constants related to cache size and sampling ratio [14]. Aggregating the cost of sorting and the cost of manipulating the data structure, we summarize the cost for compact-graph by

$$T(n,p) = \langle M_E \; ; \; T_C \rangle = \left\langle \left(4 + 2\frac{c\log(2m/p)}{\log z}\right)\frac{2m}{p} \; ; \; \mathrm{O}\left(\tfrac{2m}{p}\log 2m\right) \right\rangle . \tag{3}$$

The value of m decreases with each successive iteration dependent on the topology and edge weight assignment of the input graph. Because the number of vertices is reduced by at least half each iteration, m decreases by at least $\frac{n}{2}$ edges each iteration. For the sake of simplifying the analysis, though, we use m unchanged as the number of edges during each iteration; clearly an upper bound of the worst case. Hence, the complexity of **Bor-EL** is given as

$$T(n,p) = \langle M_E \; ; \; T_C \rangle = \left\langle \left(\frac{8m + n + n\log n}{p} + \frac{4mc\log(2m/p)}{p\log z}\right)\log n \; ; \; \mathrm{O}\left(\tfrac{m}{p}\log m \log n\right) \right\rangle . \tag{4}$$

We base this assumption on the following observation. For random sparse graphs m decreases slowly in the first several iterations of **Bor-EL** , and the graph becomes denser (as n decreases at a faster rate than m) until a certain point, m decreases drastically. Table 1 illustrates how m changes for two random sparse graphs.

	G_1 = 1,000,000 vertices, 600,006 edges				G_2 = 10,000 vertices, 30,024 edges			
iteration	$2m$	decrease	% dec.	m/n	$2m$	decrease	% dec.	m/n
1	12000012	N/A	N/A	6.0	60048	N/A	N/A	3.0
2	10498332	1501680	12.5%	21.0	44782	15266	25.4%	8.9
3	10052640	445692	4.2%	98.1	34378	10404	23.2%	33.5
4	8332722	1719918	17.2%	472.8	6376	28002	80.5%	35.0
5	1446156	6886566	82.6%	534.8	156	6220	97.6%	6.0
6	40968	1405188	97.2%	100.9	2	154	98.7%	0.5
7	756	40212	98.2%	13.5				
8	12	744	98.4%	1.5				

Table 1: Example of the rate of decrease of the number m of edges for two random sparse graphs. The $2m$ column gives the size of the edge list, the *decrease* column shows how much the size of the edge list decreases in the current iteration, the *% dec.* column gives the percentage that the size of the edge list decreases in the current iteration, and m/n shows the density of the graph.

In Table 1 for graph G_1, 8 iterations are needed for Borůvka's algorithm. Until the 4th iteration, m is still more than half of its initial value. Yet at the next iteration, m drastically reduces to about $1/10$ of its initial value. Similar behavior is also observed for G_2. As for a quite substantial number of iterations m decreases slowly, for simplicity it is reasonable to assume that m remains unchanged (an upper bound for the actual m).

Table 1 also suggests that instead of growing a spanning tree for a relatively denser graph, if we can exclude heavy edges in the early stages of the algorithm and decrease m, we may have a more efficient parallel implementation for many input graphs because we might be able to greatly reduce the size of the edge list. After all, for a graph with $m/n \geq 2$, more than half of the edges are not in the MST. In fact several MST algorithms exclude edges from the graph using the "cycle" property. Cole *et al.* [8] present a linear-work algorithm that first uses random sampling to find a spanning forest F of graph G, then identifies the heavy edges to F and excludes them from the final MST. The algorithm presented in [17], an inherently sequential procedure, also excludes edges according to the "cycle" property of MST.

Without going into the input-dependent details of how vertex degrees change as the Borůvka variants progress, we compare the complexity of the first iteration of **Bor-AL** with **Bor-EL** because in each iteration these approaches compute similar results in different ways. For **Bor-AL** the complexity of the first iteration is

$$T(n,p) = \langle M_E \; ; \; T_C \rangle = \left\langle \left(\frac{8n+5m+n\log n}{p} + \frac{2nc\log(n/p)+2mc\log(m/n)}{p\log z} \right) \; ; \; \mathrm{O}\left(\tfrac{n}{p}\log m + \tfrac{m}{p}\log(m/n) \right) \right\rangle . \tag{5}$$

While for **Bor-EL** , the complexity of the first iteration is

$$T(n,p) = \langle M_E\ ;\ T_C \rangle = \left\langle \left(\frac{8m+n+n\log n}{p} + \frac{4mc\log(2m/p)}{p\log z} \right)\ ;\ \mathrm{O}\left(\tfrac{m}{p}\log m\right) \right\rangle. \tag{6}$$

We see that **Bor-AL** is a faster algorithm than **Bor-EL** , as expected, since the input for **Bor-AL** is “bucketed” into adjacency lists, versus **Bor-EL** that is an unordered list of edges, and sorting each bucket first in **Bor-AL** saves unnecessary comparisons between edges that have no vertices in common. We can consider the complexity of **Bor-EL** then to be an upper bound of **Bor-AL** .

In **Bor-FAL** n reduces at least by half while m stays the same. Compact-graph first sorts the n vertices, then assigns $\mathrm{O}(n)$ pointers to append each vertex’s adjacency list to its supervertex’s. For each processor, sorting takes $\mathrm{O}\left(\frac{n}{p}\log n\right)$ time, and assigning pointers takes $\mathrm{O}(n/p)$ time assuming each processor gets to assign roughly the same amount of pointers. Updating the lookup table costs each processor $\mathrm{O}(n/p)$ time. As n decreases at least by half, the aggregate running time for compact-graph is:

$$T_C(n,p)_{cg} = \frac{1}{p}\sum_{i=0}^{\log n}\frac{n}{2^i}\log\frac{n}{2^i} + \frac{2}{p}\sum_{i=0}^{\log n}\frac{n}{2^i} = \mathrm{O}\left(\tfrac{n\log n}{p}\right), M_E(n,p)_{cg} \le \frac{8n}{p} + \frac{4cn\log(n/p)}{p\log z}. \tag{7}$$

With **Bor-FAL** , to find the smallest weight edge for the supervertices, all the m edges will be checked, each processor covering $\mathrm{O}(m/p)$ edges. The aggregate running time is $T_C(n,p)_{fm} = \mathrm{O}(m\log n/p)$ and the memory access complexity is $M_E(n,p)_{fm} = m/p$. For the finding connected component step, each processor takes $T_{cc} = \mathrm{O}\left(n\log\frac{n}{p}\right)$time, and $M_E(n,p)_{cc} \le 2n\log n$.

The complexity for the whole Borůvka’s algorithm is:

$$\begin{aligned} T(n,p) &= T(n,p)_{fm} + T(n,p)_{cc} + T(n,p)_{cg} \\ &\le \left\langle \frac{8n+2n\log n+m\log n}{p} + \frac{4cn\log(n/p)}{p\log z}\ ;\ \mathrm{O}\left(\tfrac{m+n}{p}\log n\right) \right\rangle \end{aligned} \tag{8}$$

It would be interesting and important to check how well our analysis and claim fit with the actual experiments. Detailed performance results are presented in Section 5, here we show that **Bor-AL** in practice runs faster than **Bor-EL** and **Bor-FAL** greatly reduces the compact-graph time. Fig. 2 shows for the three approaches the breakdown of the running time for the three steps.

Immediately we can see that for **Bor-EL** and **Bor-AL** the compact-graph step dominates the running time. **Bor-EL** takes much more time than **Bor-AL** , and only gets worse when the graphs get denser. In contrast the execution time of compact-graph step of **Bor-FAL** is greatly reduced, in the experimental section with a random graph of 1M vertices and 10M edges, it is over 50 times faster than **Bor-EL** , and over 7 times faster than **Bor-AL** . Actually the execution time of the compact-graph step of **Bor-FAL** is almost the same for the three input graphs because it only depends on the number of vertices. As predicted, the execution time of the find-min step of **Bor-FAL** increases. And the connect-components step only takes a small fraction of the execution time for all approaches.

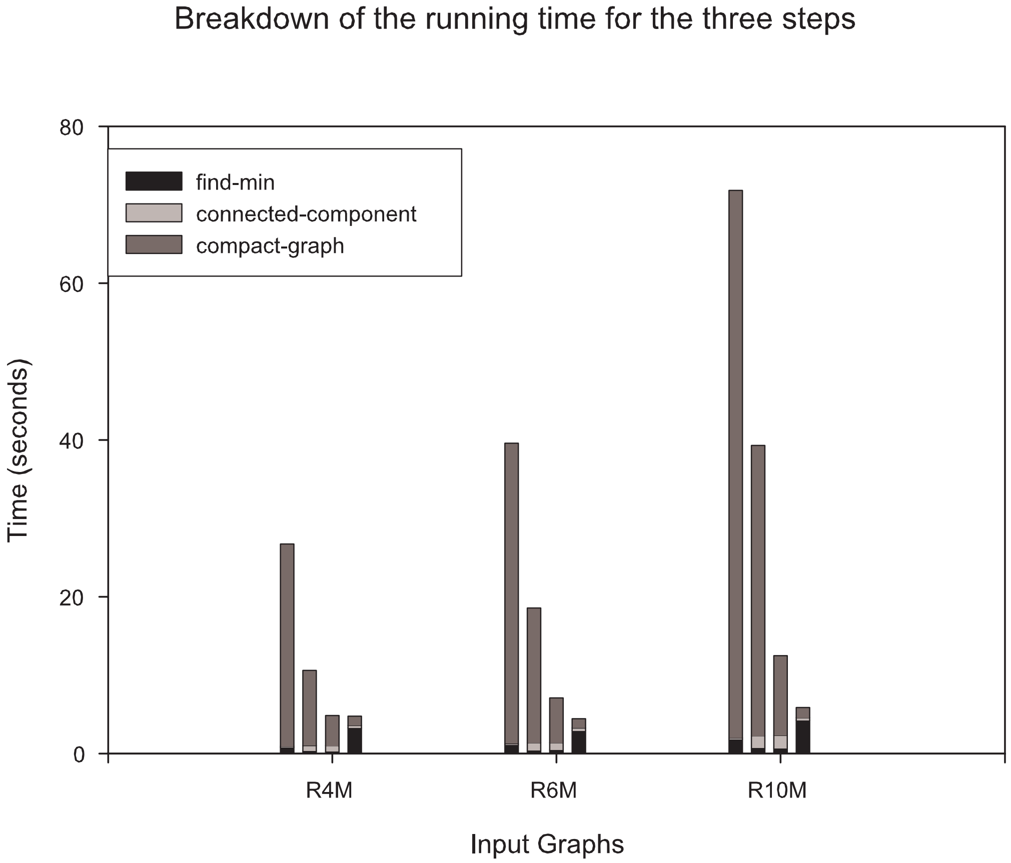


Figure 2: Breakdown of the running time for the three steps of the three approaches. Stack bars are organized into three groups by the input graphs that are random graphs with fixed 1M vertices and 4M, 6M and 10M edges respectively. Inside each group, the bars from left-to-right represent **Bor-EL** , **Bor-AL** , **Bor-ALM** , and **Bor-FAL** , respectively.

4 A New Parallel MST Algorithm For Shared-Memory

In this section we present a new non-deterministic shared-memory algorithm for finding a minimum spanning tree/forest that is quite different from Borůvka's approach in that it uses multiple, coordinated instances of Prim's sequential algorithm running on the graph's shared data structure. In fact, the new approach marries Prim's algorithm (known as an efficient sequential algorithm for MST) with that of the naturally parallel Borůvka approach. In our new algorithm essentially we let each processor simultaneously run Prim's algorithm from different starting vertices. We say a tree is *growing* when there exists a lightweight edge that connects the tree to a vertex not yet in another tree, and *mature* otherwise. When all of the vertices have been incorporated into mature subtrees, we contract each subtree into a supervertex and call the approach recursively until only one supervertex remains. When the problem size is small enough, one processor solves the remaining problem using the best sequential MST algorithm. If no edges remain between supervertices, we halt the algorithm and return the minimum spanning forest. The detailed algorithm is given in Alg. 1.

In Alg. 1, step 1 initializes each vertex as uncolored and unvisited. A processor *colors* a vertex if it is the first processor to insert it into a heap, and labels a vertex as *visited* when it is extracted from the heap; i.e., the edge associated with the vertex has been selected to be in the MST. In step 2 (Alg. 2) each processor first searches its own portion of the list for uncolored vertices from which to start Prim's algorithm. In each iteration a processor chooses a unique color (different from other processors' colors or the colors it has used before) as its own color. After extracting the minimum element from the heap, the processor checks whether the element is colored by itself, and if not, a collision with another processor occurs (meaning multiple processors try to color this element in a race) , the processor stops growing the current sub-MST. Otherwise it continues. In Appendix B we prove that Alg. 1 finds an MST of the graph.

The algorithm as given may not keep all of the processors equally busy, since each may visit a different number of vertices during an iteration. We balance the load simply by using the work stealing technique as follows. When a processor completes its partition of $\frac{n}{p}$ vertices, an unfinished partition is randomly selected, and processing begins from a decreasing pointer that marks the end of the unprocessed list. It is theoretically possible that no edges are selected for the growing trees, and hence, no progress made during an iteration of the algorithm (although this case is highly unlikely in practice). For example, if the input contained $\frac{n}{p}$ cycles, with cycle i defined as vertices $\{i\frac{n}{p},(i+1)\frac{n}{p},\ldots,(i+p-1)\frac{n}{p}\}$, for $0 \leq i < \frac{n}{p}$, and if the processors are perfectly synchronized, each vertex would be a singleton in its own mature tree. A practical solution that guarantees progress with high probability is to randomly reorder the vertex set, which can be done simply in parallel and without added asymptotic complexity [30].

4.1 Analysis

Our new parallel MST algorithm possesses an interesting feature: when run on one processor the algorithm behaves as Prim's, and on n processors becomes Borůvka's, and runs as a hybrid combination for $1 < p < n$, where p is the number of processors. In addition, our new algorithm is novel when compared with Borůvka's approach in the following ways.

```
Input: Graph $G = (V, E)$ represented by adjacency list $A$ with $n = |V|$
       $n_b$: the base problem size to be solved sequentially.
Output: MSF for graph $G$
begin
    while  $n > n_b$ and $m > n - 1$ do
        1. Initialized the color and visited arrays
        for $v \leftarrow i\frac{n}{p}$ to $(i+1)\frac{n}{p} - 1$ do
            $color[v] = 0, visited[v] = 0$
        2. Run Alg. 2.
        3. for $v \leftarrow i\frac{n}{p}$ to $(i+1)\frac{n}{p} - 1$ do
            if visited$[v] = 0$ then find the lightest incident edge $e$ to $v$, and label $e$ to be in MST
        4. With the found MST edges, run connected components on the induced graph, and shrink each component into a supervertex
        5. Set $n \leftarrow$ the number of supervertices; and $m \leftarrow$ the number of edges between the supervertices
    6. if $m > n - 1$ then solve the problem on one processor else select remaining $m$ edges
end
```

Algorithm 1: Parallel algorithm for new MSF approach, for processor p_i, for $(0 \leq i \leq p - 1)$. Assume w.l.o.g. that p divides n evenly.

```
Input: (1) $p$ processors, each with processor ID $p_i$, (2) a partition of adjacency list for each
       processor (3) array color and visited
Output: A spanning forest that is part of graph $G$'s MST
begin
  1. for $v \leftarrow i\frac{n}{p}$ to $(i+1)\frac{n}{p}-1$ do
     1.1 if color$[v] \neq 0$ then $v$ is already colored, continue
     1.2 $n = n+1$, $my\_color = np + p_i$, $color[v] = my\_color$
     1.3 insert $v$ into heap $H$
     1.4 while $H$ is not empty do
         $w = heap\_extract\_min(H)$
         if (color$[w] \neq$ my_color) OR (any neighbor $u$ of $w$ has color other than 0 or
         my_color) then break
         if visited$[w] = 0$ then
             $visited[w] = 1$, and label the corresponding edge $e$ as in MST
             for each neighbor $u$ of $w$ do
                 if color$[u] = 0$ then $color[u] = my\_color$
                 if $u$ in heap $H$ then $heap\_decrease\_key(u,h)$
                 else $heap\_insert(u,H)$
end
```

Algorithm 2: Parallel algorithm for new MST approach based on Prim's that finds parts of MST, for processor p_i, for $(0 \leq i \leq p-1)$. Assume w.l.o.g. that p divides n evenly.

1. Each of p processors in our algorithm finds for its starting vertex the smallest-weight edge, contracts that edge, and then finds the smallest-weight edge again for the contracted supervertex. We do not find all the smallest-weight edges for all vertices, synchronize, and then compact as in the parallel Borůvka's algorithm.

2. Our algorithm adapts for any number p of processors in a practical way for SMPs, where p is often much less than n, rather than in parallel implementations of Borůvka's approach that appear as PRAM emulations with p coarse-grained processors that emulate n virtual processors.

The performance of our new algorithm is dependent on its granularity $\frac{n}{p}$, for $1 \leq p \leq n$. The worst-case is when the granularity is small, i.e., a granularity of 1 when $p = n$ and the approach turns to Borůvka . Hence, the worst case complexities are similar to that of the parallel Borůvka variants analyzed previously. Yet in practice we expect our algorithm to perform better than parallel Borůvka's algorithm on sparse graphs because their lower connectivity implies that our algorithm behaves like p simultaneous copies of Prim's algorithm with some synchronization overhead.

5 Experimental Results

This section summarizes the experimental results of our implementations 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. The algorithms are implemented using POSIX threads and a library of parallel primitives developed by our group [3].

5.1 Experimental Data

Next we describe the collection of sparse graph generators that we use to compare the performance of the parallel minimum 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 [13, 21, 15, 12], the random graphs used by [13, 7, 15, 12], and the geometric graphs used by [13, 15, 21, 12, 7].

- **Regular and Irregular Meshes** Computational science applications for physics-based simulations and computer vision commonly use mesh-based graphs. All of the edge weights are uniformly random.

  - **2D Mesh** The vertices of the graph are placed on a 2D mesh, with each vertex connected to its four neighbors whenever they exist.
  - **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.

- **Structured Graphs** These graphs are used by Chung and Condon (see [7] for detailed descriptions) to study the performance of parallel Borůvka's algorithm. They have recursive structures that correspond to the iteration of Borůvka's algorithm and are degenerate (the input is already a tree).
  - **str0** At each iteration with n vertices, two vertices form a pair. So with Borůvka's algorithm, the number of vertices decrease exactly by a half in each iteration. In term of the number of iterations, **str0** is a worst case for parallel Borůvka's algorithm.
  - **str1** At each iteration with n vertices, $\sqrt{n}$ vertices form a linear chain.
  - **str2** At each iteration with n vertices, $n/2$ vertices form linear chain, and the other $n/2$ form pairs.
  - **str3** At each iteration with n vertices, $\sqrt{n}$ vertices form a complete binary tree.
- **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 [24]. The edge weights are selected uniformly and at random.
- **Geometric Graphs** In these graphs, we give each vertex a fixed degree k. Moret and Shapiro [25] use these in their empirical study of sequential MST algorithms.

5.2 Performance Results and Analysis

In this section we offer a collection of our performance results that demonstrate for the first time a parallel minimum spanning tree implementation that exhibits speedup when compared with the best sequential approach over a wide range of sparse input graphs. We implemented three sequential algorithms: Prim's algorithm with binary heap, Kruskal's algorithm with non-recursive merge sort (which in our experiments has superior performance over qsort, GNU quicksort, and recursive merge sort for large inputs) and the $m \log m$ Borůvka's algorithm.

Previous studies such as [7] compare their parallel implementations with sequential Borůvka (even though they reported that sequential Borůvka was several times slower than other MST algorithms) and Kruskal's algorithm. We observe Prim's algorithm can be 3 times faster than Kruskal's algorithm for some inputs. Density of the graphs is not the only determining factor of the performance ranking of the three sequential algorithms. Different assignment of edge weights is also important. Fig. 3 shows the performance rankings of the three sequential algorithms over a range of our input graphs.

In our performance results we specify which sequential algorithm achieves the best result for the input and use this algorithm when determining parallel speedup. In our experimental studies, **Bor-EL** , **Bor-AL** , **Bor-ALM** , and **Bor-FAL** , are the parallel Borůvka variants using edge lists, adjacency lists, adjacency lists and our memory management, and flexible adjacency lists, respectively. **MST-BC** is our new minimum spanning forest parallel algorithm.

In Appendix A we present a summary of our performance results. The performance plots in Fig. 4 are for the random graphs, in Fig. 5 are for the regular and irregular meshes (mesh, **2D60**, and **3D40**) and a geometric graph with $k = 6$, and in Fig. 6 are for the structured graphs. In these plots,

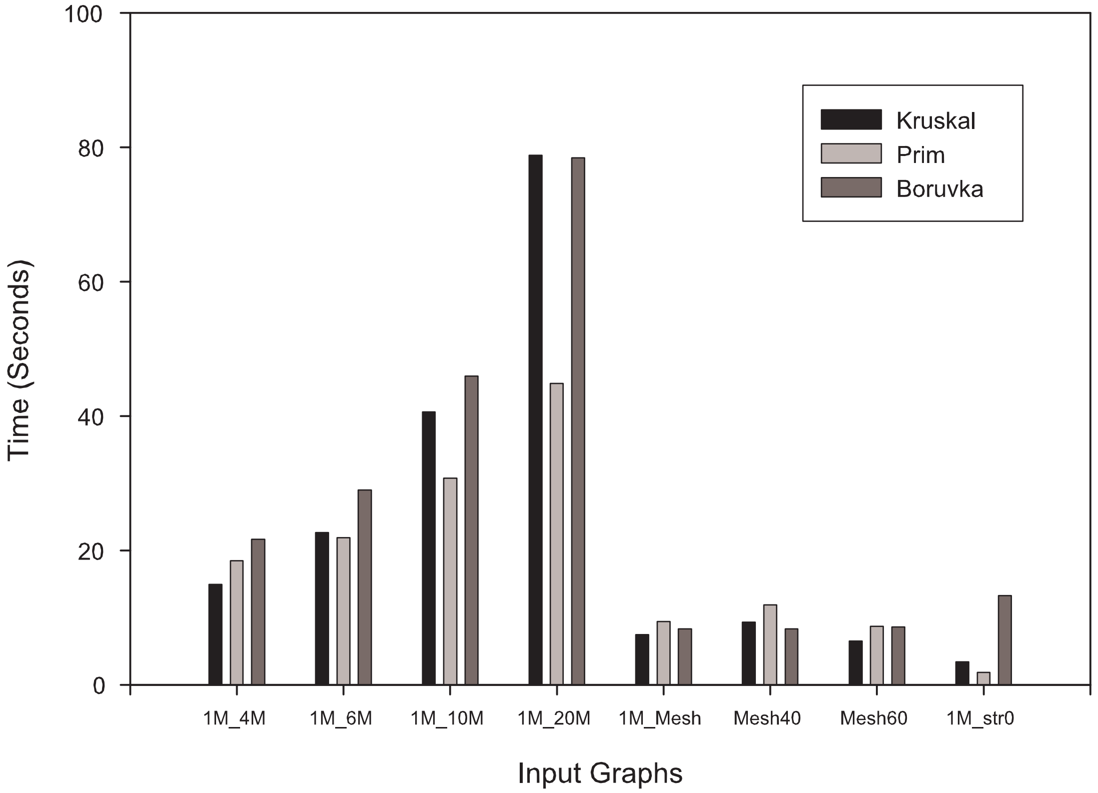


Figure 3: Different performance rankings for the three sequential algorithms over different input graph types.

the thick horizontal line represents the time taken for the best sequential MST algorithm (named in each legend) to find a solution on the same input graph using a single processor on the Sun E4500.

For the random, sparse graphs, we find that our Borůvka variant with flexible adjacency lists often has superior performance, with a speedup of approximately 5 using 8 processors over the best sequential algorithm (Prim's in this case). In the regular and irregular meshes, the adjacency list representation with our memory management (**Bor-ALM**) often is the best performing parallel approach with parallel speedups near 6 for 8 processors. Finally, for the structured graphs that are worst-cases for Borůvka algorithms, our new MST algorithm often is the only approach that runs faster than the sequential algorithm, although speedups are more modest with at most 4 for 8 processors in some instances.

6 Conclusions and Future Work

In summary, we present optimistic results that for the first time, show that parallel minimum spanning tree algorithms run efficiently on parallel symmetric multiprocessors for graphs with irregular topologies. We present a new nondeterministic MST algorithm that uses a load balancing scheme based upon work stealing that, unlike Borůvka variants, gives some speedup compared with the best sequential algorithms on several structured inputs that are hard to achieve parallel speedup. Through comparison with the best sequential implementation, we see our implementations exhibiting parallel speedup, which is remarkable to note since the sequential algorithm has very low overhead. 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, maximum flow, connected components, and planarity testing algorithms, for symmetric multiprocessors.

References

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kutylowski, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Proc. 10th Ann. Symp. Parallel Algorithms and Architectures (SPAA-98)*, pages 27–36, Newport, RI, June 1998. ACM.

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

[3] 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.

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

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

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

[7] 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.

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

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

[10] 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. co-located with the 17th IEEE Symp. on Reliable Distributed Systems.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] I. Katriel, P. Sanders, and J. L. Träff. A practical minimum spanning tree algorithm using the cycle property. Technical Report MPI-I-2002-1-003, MPI Informatik, Germany, October 2002.

[18] I. Katriel, P. Sanders, and J. L. Träff. A practical minimum spanning tree algorithm using the cycle property. In *11th Ann. European Symp. on Algorithms (ESA 2003)*, Budapest, September 2003. to appear.

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

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

[21] 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.

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

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

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

[25] 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.

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

[27] J. Park, M. Penner, and V.K. Prasanna. Optimizing graph algorithms for improved cache performance. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, Fort Lauderdale, FL, April 2002.

[28] 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.

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

[30] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–309, 1998.

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

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

A Performance Graphs

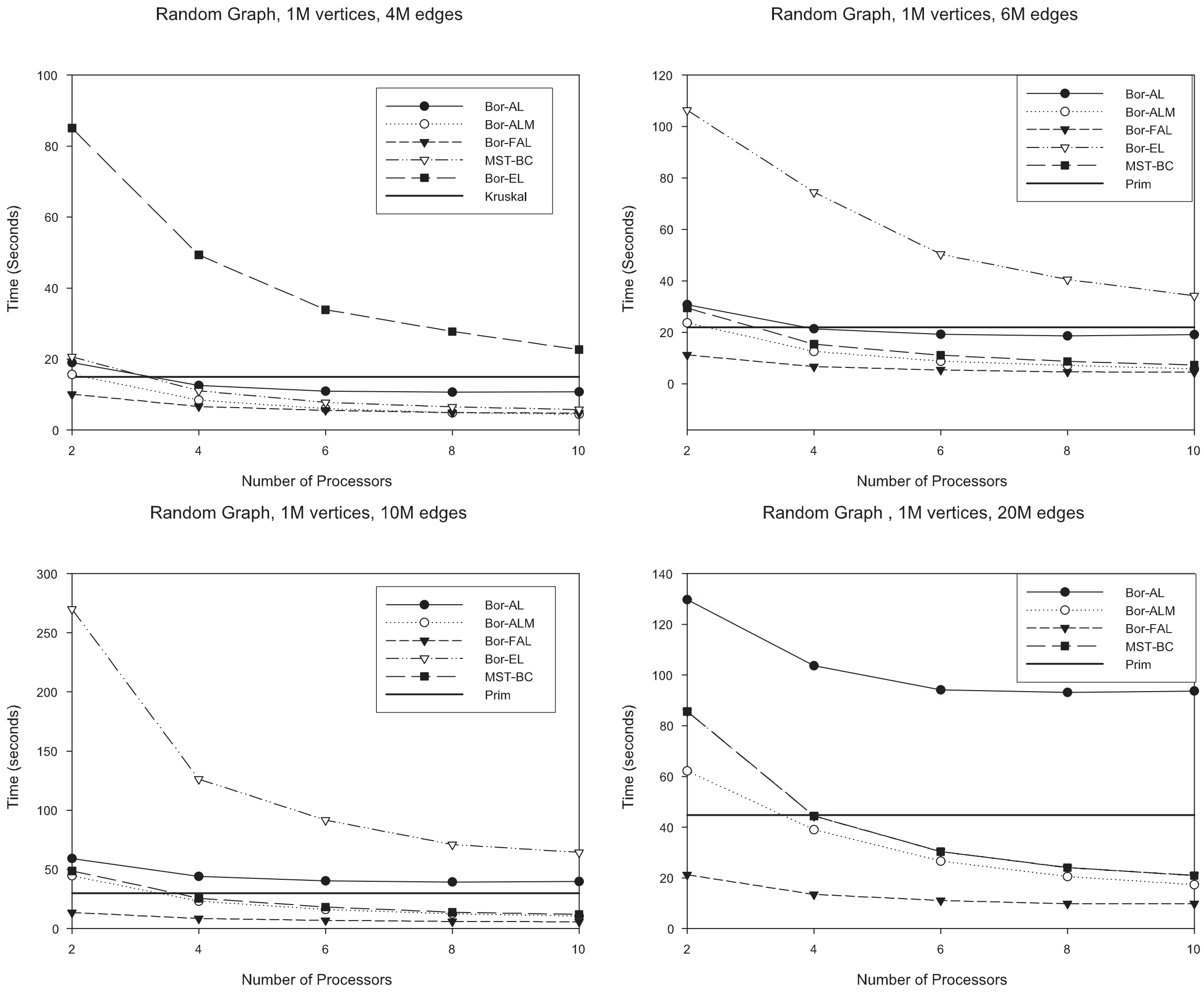


Figure 4: Comparison of parallel minimum spanning tree algorithms for random graph with $n = 1M$ vertices and $m = 4M$, $6M$, $10M$, and $20M$, edges in the top-left, top-right, bottom-left, and bottom-right, plots, respectively.

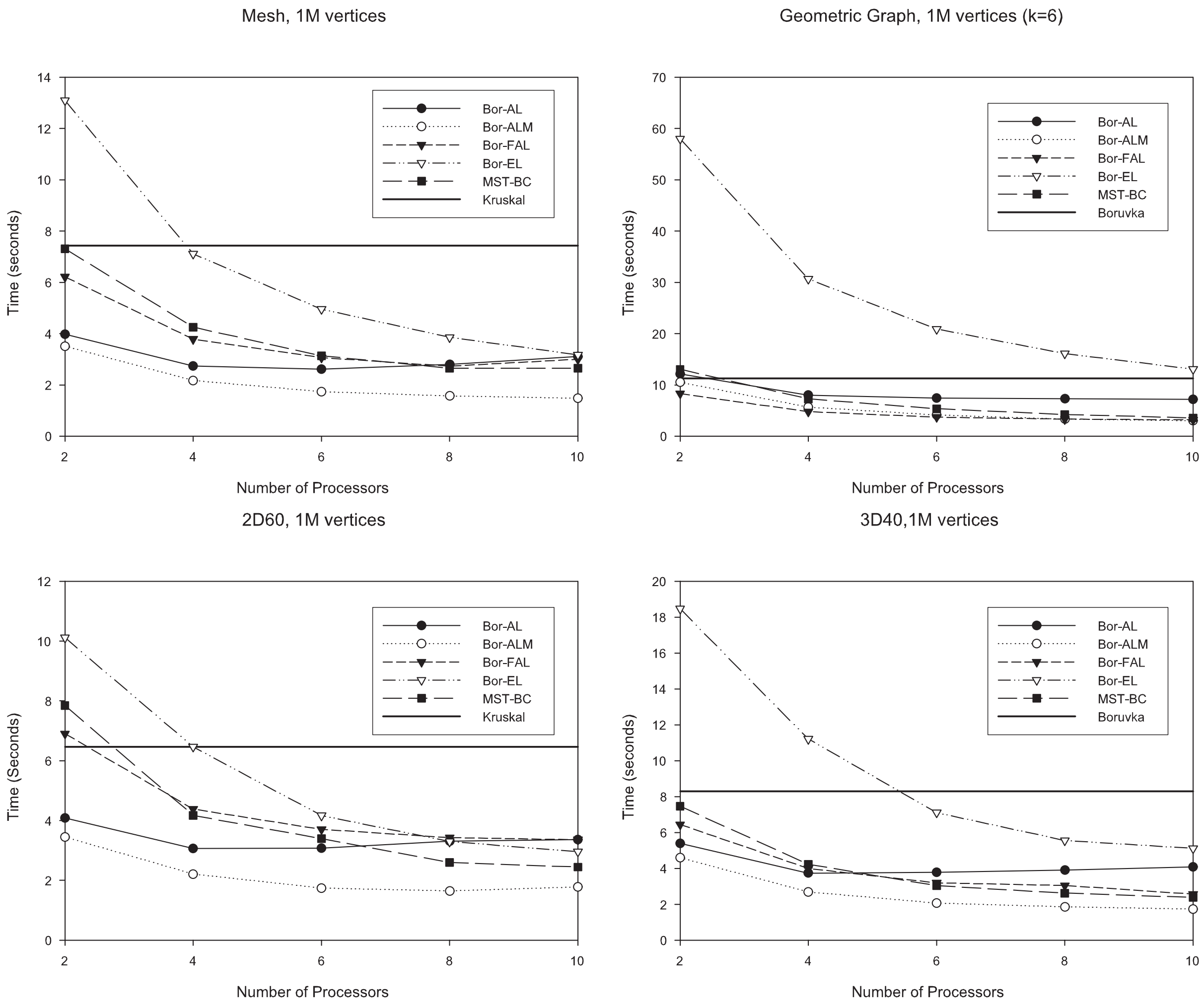


Figure 5: Comparison of parallel minimum spanning tree algorithms for regular and irregular meshes with $n = 1M$ vertices. The top-left and top-right plots show the performance of a regular mesh and a geometric graph with fixed degree $k = 6$. The bottom-left and bottom-right plots are for the **2D60** and **3D40** meshes, respectively.

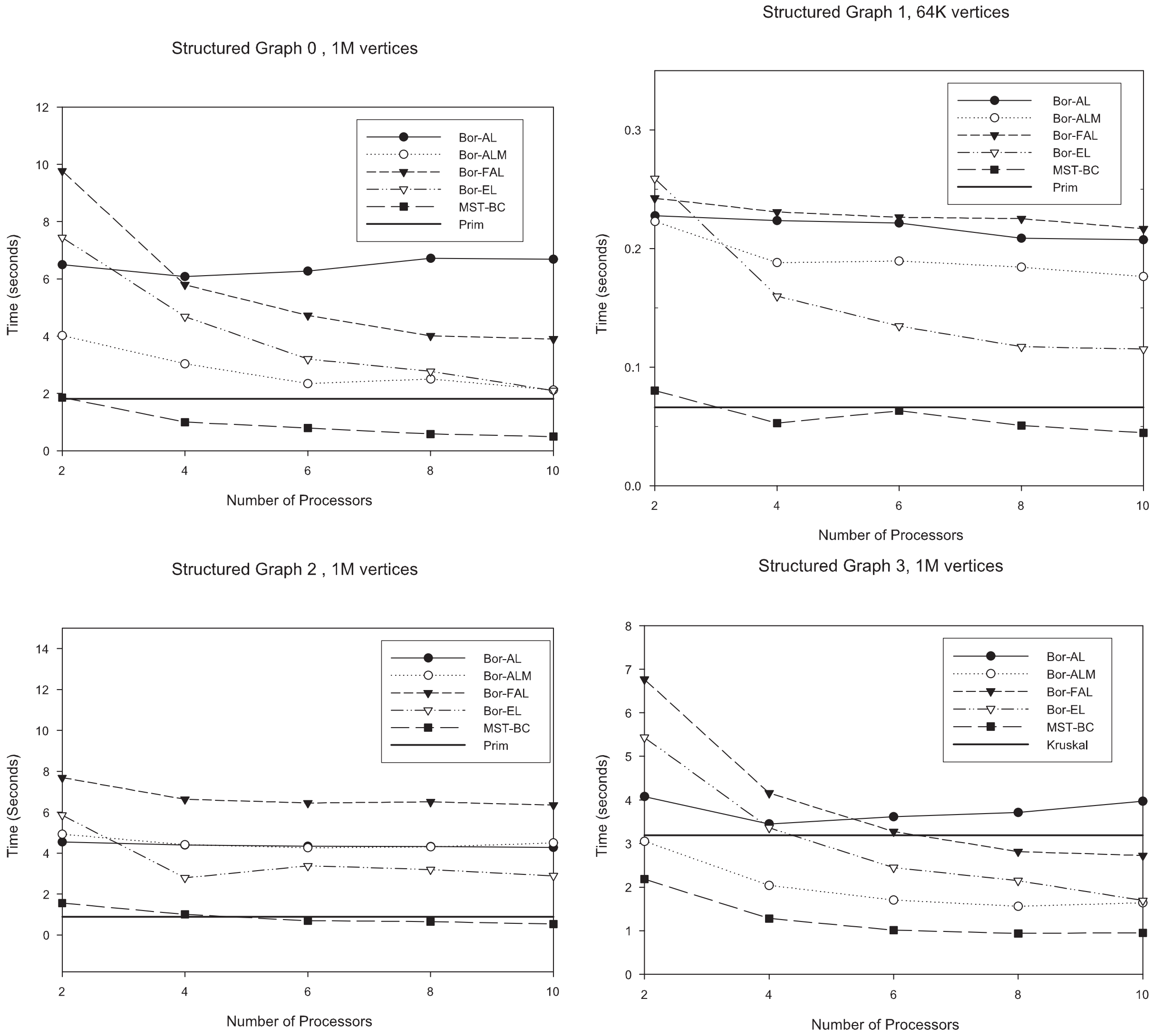


Figure 6: Comparison of parallel minimum spanning tree algorithms for the structured graphs **str0**, **str1**, **str2**, and **str3**, in the top-left, top-right, bottom-left, and bottom-right, plots, respectively.

B Proofs

We prove that Alg. 1 finds a MST of the graph. Note that we assume without loss of generality that all the edges have distinct weights.

Lemma 1 *On an SMP with sequential memory consistency, subtrees grown by Alg. 2 do not touch each other, in other words, no two subtrees share a vertex.*

Proof: Step 1.4 of Alg. 2 grows subtrees following the fashion of Prim's algorithm. Suppose two subtrees T_1 and T_2 share one vertex v. We have two cases:

- case 1: T_1 and T_2 could be two trees grown by one processor, or
- case 2: each tree is grown by a different processor.

v will be included in a processor's subtree only if when it is extracted from the heap and found to be colored as the processor's current color (Step 1.4 of Alg. 2).

(case 1) If T_1 and T_2 are grown by the same processor p_i (also assume without loss of generality T_1 is grown before T_2 in iterations k_1 and k_2 respectively with $k_1 < k_2$), and processor p_i chooses a unique color to color the vertices (Step 1.2 of Alg. 2), then v is colored k_1p+i in T_1, and later colored again in T_2 with a different color k_2p+i. As before coloring a vertex, each processor will first checks whether it has already been colored (Step 1.1 of Alg. 2), this means when processor p_i checks whether v has been colored, it does not see the previous coloring. This is a clear contradiction of sequential memory consistency.

(case 2) Assume that v is a vertex found in two trees T_1 and T_2 grown by two processors p_1 and p_2, respectively. We denote t_v as the time that v is colored. Suppose when v is added to T_1, it is connected to vertex v_1, and when it is added to T_2, it is connected to v_2. Since v is connected to v_1 and v_2, we have that $t_{v_1} < t_v$ and $t_{v_2} < t_v$. Also $t_v < t_{v_1}$ and $t_v < t_{v_2}$ since after adding v to T_1 we have not seen the coloring of v_2 yet, and similarly after adding v to T_2 we have not seen the coloring of v_1 yet. This is a contradiction of step 1.4 in Alg. 2, and hence, a vertex will not be included in more than one growing tree.

□

Lemma 2 *No cycles are formed by the edges found in Alg. 1.*

Proof: In Step 2 of Alg. 1, each processor grows sub-trees. Following lemma. 1, no cycles are formed among these trees. Step 5 of Alg. 1 is the only other step that labels edges, and the edges found in this step do not form cycles among themselves (otherwise it is a direct contradiction of the correctness of Borůvka's algorithm). Also these edges do not form any cycles with the sub-trees grown in step 2. To see this, note that each of these edges has at least one endpoint that is not shared by any of the sub-trees, so the sub-trees can be treated as "vertices." Suppose l such edges and m subtrees form a cycle, we have l edges and $l+m$ vertices, which means $m = 0$. Similarly edges found in step 5 do not connect two subtrees together, but may increase the sizes of subtrees. □

Lemma 3 *Edges found in Alg. 1 are in the MST.*

Proof: Consider a single iteration of Alg. 1 on graph G. Assume after step 5, we run parallel Borůvka's algorithm to get the minimum spanning tree for the reduced graph. Now we prove that for the spanning tree T we get from G, every edge e of G that is not in T is a T-heavy edge. Lets consider the following cases:

- Two endpoints of e are in two different subtrees. Obviously e is T-heavy because we run Borůvka's algorithm to get the minimum spanning tree of the reduced graph (in which each subtree is now a vertex) .
- Two endpoints u, v of e are in the same sub-tree that is generated by step 1.4. According to Prim's algorithm e is T-heavy.
- Two endpoints u,v of e are in the same sub-tree, u is in the part grown by step 1.4 and v is in part grown by step 3. It is easy to prove that e has larger weight than all the weights of the edges along the path from u to v in T.
- Two endpoints u, v are in the the same subtree, both u and v are in parts generated by step 5. Again e is T-heavy.

In summary, we have a spanning tree T, yet all the edges of G that are not in T are T-heavy, so T is a MST. □

Theorem 1 *For connected graph G, Alg. 1 finds the MST of G.*

Proof:

Theorem 1 follows by repeatedly applying Lemma 3. □