A slightly different version appears in IPL 62(3):153-159, 1997

An Optimal EREW PRAM Algorithm For Minimum Spanning Tree Verification

Valerie King*, Chung Keung Poon†, Vijaya Ramachandran‡ and Santanu Sinha§

July 29, 1997

Abstract

We present a deterministic parallel algorithm on the EREW PRAM model to verify a minimum spanning tree of a graph. The algorithm runs on a graph with n vertices and m edges in $O(\log n)$ time and $O(m+n)$ work. The algorithm is a parallelization of King's linear time sequential algorithm for the problem.

1 Introduction

The problem of verifying if a given spanning tree in an edge-weighted graph is a minimum spanning tree for the graph is an important problem which is closely related to the problem of finding a minimum spanning tree of a graph. Recently Dixon, Rauch and Tarjan ([DRT92]) and King ([Kin95]) have developed sequential linear time algorithms for the problem. Dixon and Tarjan have also given an optimal CREW algorithm ([DT94]). In this paper, we present a parallel algorithm which runs in optimal time and work bounds on the weaker EREW PRAM model. This resolves an open question posed in [DT94].

The high-level structure of the algorithm has been adapted from [DT94] and [Kin95]. We use tree contraction (shunting) ([KR90, KD88]) to convert the given spanning tree to a logarithmic depth *Borůvka* tree. As in [DT94] we decompose the given tree into *microtrees* of size $O(\sqrt{\log n})$. The microtrees are verified in parallel using King's ([Kin95]) sequential linear time algorithm. The process is repeated once more. This leaves a *macrotree* of size $O(n/\log n)$. This remaining macrotree is verified by forming an exhaustive query structure, which can be done in work linear in the size of the graph.

Our algorithm uses simple parallel constructs such as tree contraction and the Euler tour (see, e.g., [KR90, J´92]) along with King's ([Kin95]) algorithm as a procedure. The problem of verifying a

*Department of Computer Science, University of Victoria, Victoria, Canada. This research was supported in part by NSERC.

†Department of Computer Sciences, University of Texas at Austin, TX 78712. This research was supported in part by Texas Advanced Research Projects grant 0003658386.

‡Department of Computer Sciences, University of Texas at Austin, TX 78712. This research was supported in part by NSF grant CCR/GER 90-23059 and Texas Advanced Research Projects grant 0003658386.

§Department of Computer Sciences, University of Texas at Austin, TX 78712.

minimum spanning tree is a key sub-problem in the randomized algorithms that have been developed for finding a minimum spanning tree. In this context, our EREW algorithm is an important step towards deriving a logarithmic time, linear work EREW PRAM algorithm for finding a minimum spanning tree.

2 Previous work

Among the earliest studies on the problem of verifying minimum spanning trees was one by Tarjan ([Tar79]) who presented a sequential algorithm whose running time was superlinear by a factor of $\alpha(m, n)$, the inverse Ackerman function. Komlós's algorithm ([Kóm85]) was the first to use a linear number of comparisons, but no implementation was known for some time. Then Dixon, Rauch and Tarjan ([DRT92]) developed a linear time algorithm based on Komlós's idea and a table look-up method. A simpler algorithm was developed by King ([Kin95]), which is essentially a simplification and implementation of the Komlós algorithm.

For parallel algorithms, Alon and Schieber ([AS87]) gave one on the CREW PRAM model whose work bound is superlinear by a factor of $\alpha(m, n)$. Parallelizing the algorithm in [DRT92], Dixon and Tarjan ([DT94]) obtained an algorithm on the CREW PRAM with optimal bounds (i.e., logarithmic time and linear work). We further improve it by parallelizing King's algorithm ([Kin95]). Our result is an optimal algorithm for this problem on the EREW PRAM, which is the weakest PRAM model. All of the above algorithms are deterministic.

It should be mentioned that an algorithm for finding a minimum spanning tree is also an algorithm for verification. Among the known parallel algorithms are Chong and Lam's deterministic EREW algorithm which requires $O(\log n \log\log n)$ time on $O(m + n)$ processors ([CL93]), Karger's randomized EREW algorithm which runs in $O(\log n)$ expected time on $\Theta((m/\log n) + n^{1+\epsilon})$ processors ([Kar95]) and Cole, Klein and Tarjan's randomized CRCW algorithm which requires logarithmic time and linear work ([CKT96]).

3 The Algorithm

Given a graph G with n vertices and m edges with a distinct weight assigned to each edge and a spanning tree T on G, we are required to verify that T is the minimum spanning tree (MST) of G. We assume distinct edge-weights without loss of generality to simplify our presentation.

The algorithm is based on the fact that a spanning tree is the minimum spanning tree if and only if the weight of each nontree edge $<u, v>$ in G is at least the weight of the heaviest edge on the path between u and v in the tree. In other words, we must verify that each nontree edge is heavier than all edges on the path in the tree it *covers*.

3.1 The Modified Boruvka Tree

Let T be a spanning tree of a connected graph G with n nodes. The *Borůvka* tree B (as defined in [Kin95]) of T is the tree of components that are formed when the Borůvka algorithm ([Bor26]) for finding a minimum spanning tree is applied to T. The Borůvka algorithm contracts the minimum edge incident on each component in every phase until there is only one component remaining.

More formally, the construction of the Borůvka tree starts with a tree $T = (V, E)$, and creates a leaf $f(v)$ of B for every vertex $v \in V$. Each vertex v is considered to be a separate component. In each phase of the Borůvka algorithm, each component chooses the minimum weight edge incident on it. The chosen edges are contracted, hence merging groups of components into single components. Each new component t is added as a new node $f(t)$ to B. $f(t)$ is connected to the sub-component nodes from which it is formed by edges of weights equal to the weights of the edges which have been contracted in T.

In our algorithm, we shall construct a *modified* Borůvka tree, which is constructed exactly as before except that in each phase, no more than a constant number of components may be merged into one. In each phase of the computation, only a subset of the chosen edges is contracted so that any node in the Borůvka tree has no more than a constant number of children.

For any tree T, let $T(x, y)$ denote the path from x to y in T and let $w(e)$ be the weight of edge e. We shall prove the following property of a modified Borůvka tree. This was proved earlier for the Borůvka tree in [Kin95].

Lemma 1 *Let T be a spanning tree and let B be a modified Borůvka tree constructed as described above. Then for any pair of nodes x and y in T, the weight of the heaviest edge in T(x,y) equals the weight of the heaviest edge in B(f(x),f(y)).*

Proof: First we show that for every edge $e \in B(f(x), f(y))$, there exists an edge $e' \in T(x, y)$ such that $w(e') \geq w(e)$.

Let $e = <a, b>$, and let a be the endpoint of e which is farther away from the root of B. Then, $a = f(t)$, where t is a component that contains either x or y, but not both. $w(e)$ is the weight of the edge selected by t to be contracted. Let e' be the edge in $T(x, y)$ with exactly one endpoint in t. Since t must choose the minimum weight edge among e and e' and it chooses e, we have $w(e') \geq w(e)$.

Let e be the heaviest edge in $T(x, y)$. It remains to show that there is an edge of the same weight in $B(f(x), f(y))$.

If e is selected for contraction by a component which contains x or y, then an edge in $B(f(x), f(y))$ is labeled $w(e)$. We assume the contrary. Then, e is selected for contraction by a component t which does not contain x or y. Since t contains one endpoint of e, it contains one or more intermediate nodes in $T(x, y)$. Therefore at least two edges in $T(x, y)$ are incident on t, yet the heavier edge e is selected, giving a contradiction.

■

Let T be the given spanning tree in a graph G. From Lemma 1, it follows that to verify that T is the MST of G, it is sufficient to verify that for any nontree edge $<u, v>$, the heaviest edge in $B(f(u), f(v))$ is no heavier than $w(u, v)$.

3.2 Algorithm Description

We are now ready to present our parallel algorithm. We are given an edge-weighted graph G with spanning tree T. Nontree edges are the edges in G that are not in T.

3.2.1 Formation of the modified Boruvka tree

We first *binarize* the tree T. A dummy child node is added to nodes with only one child. Nodes with $k(\geq 3)$ children are split into two sets of sizes $\left\lceil \frac{k}{2} \right\rceil$ and $\left\lfloor \frac{k}{2} \right\rfloor$ that become children of the node. The new edges are given a cost of $-\infty$, so that they are not heavier than any nontree edge. This procedure is recursively repeated until we have a binary tree. Note that the number of dummy nodes is linear in the size of the original tree.

Next, we use tree contraction as described in [KR90, KD88]. Tree contraction shrinks a tree into a single vertex by repeatedly applying the *shunt* operation (which is called *rake* in [KD88]; we prefer the term *shunt* since this operation is derived from the *shunt* operation on DAGs presented in [MRK88]). Given a leaf u whose parent is $p(u)$, the shunt operation applied to u removes u and $p(u)$ from the tree and connects the sibling of u to $p(p(u))$, the parent of $p(u)$. The tree contraction algorithm is applicable only to binary trees. The leaves are first numbered from left to right, excluding the leftmost and rightmost leaves of the tree. For $\log n$ iterations, where n is the number of vertices, the shunt operation is applied to the odd numbered leaves remaining in the tree. The shunt is applied in parallel to the odd-numbered leaves which are left children of their parents first and then applied in parallel to the odd-numbered right children. This ensures that there are no read-write conflicts. Each complete iteration reduces the number of vertices by half. Hence $O(\log n)$ iterations suffice to contract the tree into a single vertex. At the end of each iteration, only the even numbered leaves remain, so renumbering them simply involves dividing the number for each leaf by 2.

We shall apply this tree contraction to form a modified Borůvka tree. The shunt operation is performed as follows :

- Merge leaf v with its parent $p(v)$. Note that this is a valid Borůvka merge since a leaf has only one edge incident on it, which must be the minimum edge.
- Contract the minimum edge above or below $p(v)$. By the previous step, v has been removed. Hence, $p(v)$ has two edges, one to v's sibling and one to its parent. We contract the lighter of these two edges, hence maintaining the requirements for a Borůvka tree.

It should be pointed out that performing a shunt as above may result in access conflicts in a case where both v and $p(p(v))$ try to merge with $p(v)$ in the second substep. However, since the number of processors in conflict is constant, the conflicts may be resolved on the EREW PRAM without increasing time or work requirements.

The modified Borůvka tree B is formed in $O(\log n)$ iterations as in the general tree contraction algorithm. Every iteration gives rise to two levels of B, one for each parallel application of the shunt operations. Hence, B has depth $O(\log n)$. The number of children of a node of B is no more than five, since one parallel application of shunt operations can merge at most five nodes into the same component.

Henceforth, we shall refer to the modified Borůvka tree as the Borůvka tree.

3.2.2 Pre-processing of the Boruvka tree and nontree edges

Every nontree edge $<u, v>$ in G is transformed into a nontree edge $<f(u), f(v)>$ in the Borůvka tree B, which we call the corresponding *query* edge of $< u, v >$. We must now verify that $w(u, v)$ is at least as large as the weight of any edge in $B(f(u), f(v))$. For this we construct an Euler tour ([J´92, TV85]) on B and use it to determine the level, parent and size of every node. The *size* of a node is the number of nodes in its subtree including itself. Note that all query edges in B are incident only on leaves of B.

Next, for each nontree edge, we find the least common ancestor (lca) of its endpoints in B. This can be done in logarithmic time and linear work on an EREW PRAM using the preprocessing algorithm described in [J´92], together with a scheme described in [Ram96]. Then, we split each nontree edge $<x, y>$ into two edges, namely $<x, lca(x, y)>$ and $<y, lca(x, y)>$ with the same weight as $<x, y>$. These new edges can be stored at x and y respectively to avoid write conflicts. It is sufficient to verify that these new edges are heavier than the tree paths they cover. Here onwards, the nontree edges in B can only be between a leaf and its ancestor.

3.2.3 Microtrees : Cutting and Verification

For any node v in the tree B, if $size(v) \leq \sqrt{\log n} < size(p(v))$, we designate the subtree rooted at v as a *microtree*. Here, we can avoid a concurrent read of parent sizes by having parent nodes broadcasting their size to their children. Any such operation can be performed in constant time because the number of children of a node is no more than a constant.

If a nontree edge $<u, x>$ originating at a leaf u has the other endpoint x above the root r of the microtree in which u is contained, it is split into two edges $<u, r>$ and $<r, x>$ of the same weight. Edge $<u, r>$ is stored at u while $<r, x>$ is saved at r. Write conflicts at r can be avoided by a post-order numbering of the nontree edges originating from the microtree of r. Now we have two sets of edges, one consisting of edges within the microtrees and the other consisting of edges in the tree formed by removing all the microtrees. We shall call the latter tree a *macrotree*.

We can now verify in parallel that the edges within the microtrees do not violate the minimality condition, using the sequential algorithm in [Kin95] for each microtree. Processors can be allocated by computing prefix sums on an auxiliary array containing the number of nontree edges lying wholly within each microtree and making a proportional allocation of microtrees to processors. Once this step is over, the microtrees can be removed, leaving us with a tree of size $O(n/\sqrt{\log n})$.

We shall perform another step of forming microtrees, edge-splitting, parallel verification and microtree removal on the remaining tree. This step is performed exactly as before. Now, we will have reduced the remaining macrotree to size $O(n/\log n)$.

3.2.4 The Macrotree : Verification

The final stage of the verification is verifying the minimality condition for the macrotree M. The algorithm follows :

```
Pfor all nodes v do
          CURRENT(v) := $w(v, p(v))$
Rofp
For i=1 to DEPTH(M) do
          Pfor all nodes v do
                    Send CURRENT(v) to children
                    Set CURRENT(v) := New value received from parent
                    If v is a leaf then
                              Save CURRENT in Path_To_Anc[v,i]
          Rofp
Rof
Pfor all nontree edges <u,v>, where u is a leaf do
          Place edge in bucket[u,level(v)]
Rofp
```

```
Pfor all leaves v do
    Perform a prefix max on Path_To_Anc[v] array
    Pfor all buckets at v do
        Find minimum weight edge at bucket[v,i]
    Rofp
    Verify minimum weight at bucket[v,i] $\geq$ Path_To_Anc[v,i]
Rofp
```

The first part of the algorithm simply propagates edge weights to the leaves. Now, we collect all the nontree edges from a leaf and bucket sort them by level of the ancestor to which the nontree edge connects. All edges in a bucket start and end at the same nodes. So, only the minimum weight edge in each bucket is saved. The prefix max on the array *Path_To_Anc* computes the heaviest tree edge weight on the path from the leaf to any ancestor. Now we can compare the minimum values in the buckets to the corresponding value in the array. If none of the nontree edges are lighter than their corresponding entries in *Path_To_Anc* we have verified that the given spanning tree is indeed a minimum spanning tree.

3.3 Analysis

In this section, we discuss the correctness of the algorithm and analyze the running time and work required for each step of the algorithm.

3.3.1 Correctness

The algorithm is based on the well known fact that for any cycle C in a graph, the heaviest edge in C does not appear in the minimum spanning tree. We verify that for each nontree edge $<u, v>$, the weight of the edge is heavier than any edge in $T(u, v)$. It is sufficient to do the same on the modified Borůvka tree by Lemma 1. Additionally, nontree edges from T can only be between leaves of B. We say the edge $<u, v>$ *covers* the tree-path $T(u, v)$.

An edge $<u, v>$ is split into two edges, $<u, x>$ and $<v, x>$, each of which covers the tree path from one endpoint of the edge to the lowest common ancestor x of u and v. Each such edge is split into upto three parts in two levels of microtrees and the macrotree. In each microtree and in the macrotree, we have shown how to verify that each nontree edge is at least as heavy as every edge on the tree-path it covers.

3.3.2 Work and Time bounds

In the following we analyze the work and time bounds for the different steps in the algorithm.

1. (Section 3.2.1.) Binarization of the tree is performed by allocating processors to nodes according to the number of children at that node. A node with k children will require no more than $\log k$ steps for the children to be recursively split into a binary subtree. The number of nodes added in this subtree will be $k/2$. Hence, this operation can be performed in $O(\log n)$ time and linear work.

The tree contraction procedure is a logarithmic time, linear work procedure on an EREW PRAM ([KD88]). Note that each shunt operation on a leaf requires only a constant amount of work.

2. (Section 3.2.2.) In the pre-processing step, we first construct an Euler tour and compute prefix sums for parameters such as the level, size and parent of a node. This is a logarithmic time, linear work EREW computation ([TV85]).

   The least common ancestors (lca's) may be computed in the same work-time bounds with the method for computing lca's for nontree edges given in [Ram96]. (Briefly, this method is based on the algorithm for finding lca's based on the Euler tour (see, e.g., [J´92]) that preprocesses in $O(\log n)$ time and $O(n)$ work on the EREW PRAM and answers queries in constant time in parallel on the CREW PRAM. [Ram96] observes that the lca's of any collection of nontree edges can be found in $O(\log n)$ time and linear work on an EREW PRAM by this method, provided all endpoints of edges are distinct. To enforce this, [Ram96] transforms the tree by appending a chain to each vertex, with one copy of the vertex for each nontree edge incident on it, computes lca's of nontree edges in this new tree, and determines the lca's of the nontree edges in the original tree from this. Thus both the preprocessing and query steps are performed in $O(\log n)$ time and $O(m)$ work on an EREW PRAM.)

   Each nontree edge is now split into two edges. This is a constant time operation with linear work on the number of edges.

3. (Section 3.2.3.) The cutting of microtrees and splitting edges can be performed in $O(1)$ time with $O(n+m)$ processors. We shall do it in $O(\log n)$ time with $O((m+n)/\log n)$ processors. There is an implicit processor allocation step here to list the nodes and query edges, compact the list into an auxiliary array, and perform a prefix sums computation on this array to determine the allocation of tasks to processors. These steps are all within $O(\log n)$ time and $O(m+n)$ work.

   The sequential minimum spanning tree algorithm is a linear time algorithm. Hence we can perform the verification of the microtrees in parallel in $O(m+n)$ work. Since each tree has $O(\sqrt{\log n})$ vertices, it has $O(\log n)$ nontree edges within it and hence size $O(\log n)$, and we can perform this step in $O(\log n)$ time with $O((m+n)/\log n)$ processors. This includes a computation for allocating processors so that each processor is assigned a collection of microtrees of total size $\Theta(\log n)$. This processor allocation step involves compaction and prefix sums computation on the list of microtree nodes and edges within microtrees.

4. (Section 3.2.4.) The first step in this algorithm performs $O(\log n)$ iterations over all the nodes. Since the size of the macrotree has been reduced to $O(n/\log n)$, this is a linear work step.

   Prefix sums is as usual an operation within our bounds.

   The bucket sort of nontree edges on the macrotree leaves can be performed in $O(\log m)$ time using $O(m/\log m)$ processors, simply by assigning $O(\log m)$ edges to each processor, which puts each edge in its bucket. Since we have $O(m/\log m)$ processors, we can sort the buckets for minimum in $O(\log m)$ time as well. The final comparison is a constant time linear work operation.

By the above analysis it follows that the overall algorithm runs in $O(\log n)$ time and $O(m+n)$ work. These bounds are strongly optimal (as defined in [J´92]) even on a CREW PRAM. The problem of computing the OR of n bits can be mapped onto a graph with $V = \{v_1, v_2 \cdots v_n\}$, a spanning tree with edges with $<v_i, v_{i+1}>$ for $1 \leq i < n$, the weight of the i-th edge being $n+i$ if the i-th bit of the OR problem is a 1 and $n-i$ otherwise and a nontree edge $<v_1, v_n>$ of weight n. Hence, the time required by any minimum spanning tree verification algorithm on a CREW PRAM with any number of processors is $\Omega(\log n)$ ([CDR86]).

4 Conclusion

We have presented an EREW PRAM algorithm for the problem of verifying a minimum spanning tree that runs in logarithmic time and linear work. This resolves an open questions in [DT94]. The algorithm is also strongly optimal. Additionally, our algorithm has a very simple high-level structure and makes use of elementary parallel constructs.

A closely related problem is to *find* a minimum spanning tree in a graph. Cole, Klein and Tarjan have given a randomized logarithmic time, linear work CRCW PRAM algorithm for this problem ([CKT96]). This algorithm makes use of a minimum spanning tree verification algorithm as a procedure. An important open problem is to develop a logarithmic time, linear work algorithm to find a MST on the EREW or QRQW PRAM ([GMR94]) models, since these models are a better match to existing parallel machines. Our EREW verification algorithm can be used directly on any PRAM model and hence may prove useful in deriving such algorithms. As a first step to solving this open problem two of the authors of this paper have developed a linear work EREW PRAM algorithm to find a minimum spanning tree that runs in slightly super-logarithmic time [PR97]; this gives the first linear work, polylogarithmic time algorithm on the EREW PRAM for this problem.

References

[AS87] N. Alon and B. Schieber. Optimal preprocessing for answering online product queries. Technical report, Tel Aviv University, 1987.

[Bor26] O. Borůvka. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 3:37–58, 1926. In Czech.

[CDR86] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, February 1986.

[CKT96] R. Cole, P.N. Klein, and R.E. Tarjan. Finding minimum spanning trees in logarithmic time and linear work using random sampling. In *Proceedings of the 1996 ACM Symposium on Parallel Algorithms and Architectures*, pages 243–249, 1996.

[CL93] K. W. Chong and T. W. Lam. Connected components in $O(\log n \log\log n)$ time on the EREW PRAM. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–20, 1993.

[DRT92] B. Dixon, M. Rauch, and R. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[DT94] B. Dixon and R.E. Tarjan. Optimal parallel verification of minimum spanning trees in logarithmic time. In *Parallel and Distributed Computing, Theory and Practice, Lecture Notes in Compututer Science 805*, pages 13–22. Springer-Verlag, Berlin, 1994.

[GMR94] P.B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648, 1994.

[HZ96] S. Halperin and U Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *Proceedings of the Seventh ACM-SIAM Symposium on Discrete Algorithms*, pages 438–447, 1996.

[J´92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[Kar95] D. R. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Department of Computer Science, Stanford University, 1995.

[KD88] S. R. Kosaraju and A. Delcher. Optimal parallel evaluation of tree-structured computations raking. In *Aegean Workshop on Computing*, pages 101–110, 1988.

[Kin95] V. King. A simpler minimum spanning tree verification algorithm. In *Lecture Notes in Compututer Science 955*, pages 440–448. Springer-Verlag, Berlin, 1995.

[Kóm85] J. Kómlos. Linear verification for spanning trees. *Combinatorica*, 5:57–65, 1985.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoritical Computer science*, volume A:Algorithms and Complexity, pages 869–941. MIT Press, Cambridge Mass., 1990.

[MRK88] G.L. Miller, V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17(4):687–695, 1988.

[PR97] C. K. Poon and Vijaya Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning tree. In *Proceedings of the 8th Annual International Symposium on Algorithms and Computation (ISAAC'97)*, 1997. To appear.

[Ram96] V. Ramachandran. Private communication to Uri Zwick, January, 1996. To be included in journal version of [HZ96].

[Tar79] R. E. Tarjan. Applications of path compressions on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[TV85] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, November 1985.