A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest

Seth Pettie and Vijaya Ramachandran
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
seth@cs.utexas.edu, vlr@cs.utexas.edu

April 27, 1999

UTCS Technical Report TR99-13

Abstract

We present a randomized algorithm to find a minimum spanning forest (MSF) in an undirected graph. With high probability, the algorithm runs in logarithmic time and linear work on an EREW PRAM. This result is optimal with respect to both work and parallel time, and is the first provably optimal parallel algorithm for this problem under both measures.

1 Introduction

We present a randomized parallel algorithm to find a minimum spanning forest (MSF) in an edge-weighted, undirected graph. On an EREW PRAM [KR90] our algorithm runs in expected logarithmic time and linear work in the size of the input; these bounds also hold with high probability in the size of the input. This result is optimal with respect to both work and parallel time, and is the first provably optimal parallel algorithm for this problem under both measures.

Here is a brief summary of related results. Following the linear-time sequential MSF algorithm of Karger, Klein and Tarjan [KKT95] (and building on it) came linear-work parallel MST algorithms for the CRCW PRAM [CKT94, CKT96] and the EREW PRAM [PR97]. The best CRCW PRAM algorithm known to date [CKT96] runs in logarithmic time and linear work, but the time bound is not known to be optimal. The best EREW PRAM algorithm known prior to our work is the result of Poon and Ramachandran which runs in $O(\log n \log\log n 2^{\log^* n})$ time and linear work. All of these algorithms are randomized. Recently a deterministic EREW PRAM algorithm for MSF was given in [CHL99], which runs in logarithmic time with a linear number of processors, and hence with work $O((m+n)\log n)$, where n and m are the number of vertices and edges in the input graph. It was observed by Poon and Ramachandran [PR98] that the algorithm in [PR97] could be speeded up to run in $O(\log n \cdot 2^{\log^* n})$ time and linear work by using the algorithm in [CHL99] as a subroutine (and by modifying the 'Contract' subroutine in [PR97]).

In this paper we improve on the running time of the algorithm in [PR97, PR98] to $O(\log n)$, which is the best possible, to within a constant factor, and we improve on the algorithm in [CKT96] by achieving the logarithmic time bound on the less powerful EREW PRAM.

The structure of our algorithm is fairly simple. The most complex portion of our algorithm is the subroutine calls it makes to the 'CHL algorithm' for MSF [CHL99] (which we use as a black-box). As a result our algorithm can be used as a simpler alternative to several other parallel algorithms.

1. For the CRCW PRAM we can replace the calls to the CHL algorithm by calls to a simple logarithmic time, linear-processor CRCW algorithm such as the one in [AS87]. The resulting algorithm runs in logarithmic time and linear work and is considerably simpler than the MSF algorithm in [CKT96].

2. As modified for the CRCW PRAM, our algorithm is simpler than the linear-work logarithmic-time CRCW algorithm for connected components given in [Gaz91].

3. Our algorithm improves on the EREW connectivity and spanning tree algorithms in [HZ94, HZ96] since we compute a *minimum* spanning tree within the same time and work bounds. Our algorithm is arguably simpler than the algorithms in [HZ94, HZ96].

The rest of this paper describes and analyzes our algorithm. In the following we use the notation $S+T$ to denote union of sets S and T, and we use $S+e$ to denote the set formed by adding the element e to the set S. We say that a result holds *with high probability (or w.h.p.) in n* if the probability that it fails to hold is less than $1/n^c$, for any constant $c>0$.

2 The High-Level Algorithm

Our algorithm is divided into two phases along the lines of the CRCW PRAM algorithm of [CKT96]. In Phase 1, the algorithm reduces the number of vertices in the graph from n to n/k vertices, where n is the number of vertices in the input graph, and $k = (\log^{(2)} n)^2$.[†] To perform this reduction

[†]We use $\log^{(r)} n$ to denote the log function iterated r times, and $\log^* n$ to denote the minimum r s.t. $\log^{(r)} n \le 1$.

the algorithm uses the familiar recursion tree of depth $\log^* n$ [CKT94, CKT96, PR97], which gives rise to $O(2^{\log^* n})$ recursive calls, but the time needed per invocation in our algorithm is well below $O(\log n / 2^{\log^* n})$. Thus the total time for Phase 1 is $O(\log n)$. We accomplish this by requiring Phase 1 to find only a subset of the MSF. By contracting this subset of the MSF we obtain a graph with $O(n/k)$ vertices. Phase 2 then uses an algorithm similar to the one in [PR97], but needs no recursion due to the reduced number of vertices in the graph. Thus Phase 2 is able to find the MSF of the contracted graph in $O(\log n)$ time and linear work.

We assume that edge weights are unique. As always, uniqueness can be forced by ordering the vertices, then ordering identically weighted edges by their end points.

Here is a high-level description of our algorithm.

```
High-Level($G$)
    (Phase 1)  $G_t$ := For all $v \in G$, retain the lightest $k$ edges in edge-list($v$)
               $M$ := Find-k-Min($G_t, \log^* n$)
               $G'$:=Contract all edges in $G$ appearing in $M$
    (Phase 2)  $G_s$:=Sample edges of $G'$ with prob. $1/\sqrt{k} = 1/\log^{(2)} n$
               $F_s$ :=Find-MSF($G_s$)
               $G_f$ := Filter($G', F_s$)
               $F$ :=Find-MSF($G_f$)
               Return($M + F$)
```

Theorem 2.1 *With high probability, High-Level(G) returns the MSF of G in $O(\log n)$ time using $(m+n)/\log n$ processors.*

In the following sections we describe and analyze the algorithms for Phase 1 and Phase 2, and then present the proof of the main theorem for the expected running time. We then obtain a high probability bound for the running time and work. When analyzing the performance of the algorithms in Phase 1 and Phase 2, we use a time-work framework, assuming perfect processor allocation. This can be achieved with high probability to within a constant factor, using the load-balancing scheme in [HZ94], which requires superlinear space, or the linear-space scheme claimed in [HZ96]. We discuss processor allocation in Section 7 where we point out that a simple scheme similar to the one in [HZ94] takes only linear space on the *QRQW* PRAM [GMR94], which is a slightly stronger model than the EREW PRAM. The usefulness of the QRQW PRAM lies in the fact the algorithms designed on that model map on to general-purpose models such as QSM [GMR97] and BSP [Val90] just as well as the EREW PRAM. We then describe the performance of our MSF algorithm on the QSM and BSP.

3 Phase 1

In Phase 1, our goal is to contract the input graph G into a graph with $O(n/k)$ vertices. We do this by identifying certain edges in the minimum spanning forest of G and contracting the connected components formed by these edges. The challenge here is to identify these edges in logarithmic time and linear work.

Phase 1 achieves the desired reduction in the number of vertices by constructing a *k-Min forest* (defined below). This is similar to the algorithm in [CKT96]. However, our algorithm is considerably simpler. We show that a k-Min forest satisfies certain properties, and we exploit these properties to design a procedure *Borůvka-A*, which keeps the sizes of the trees contracted in the various stages of Phase 1 to be very small so that the total time needed for contracting and processing edges in these trees is $o(\log n / 2^{\log^* n})$. Phase 1 also needs a *Filter* subroutine, which removes 'k-min light' edges. We show that we can use an MSF verification algorithm on the small trees we construct to perform this step. The overall algorithm for Phase 1, Find-k-Min uses these two subroutines to achieve the stated reduction in the number of vertices within the desired time and work bounds.

3.1 k-Min Forest

Phase 1 uses the familiar 'sample, contract and discard edges' framework of earlier randomized algorithms for the MSF problem [KKT95, CKT94, CKT96, PR97]. However, instead of computing a minimum spanning forest, we will construct the *k-Min tree* [CKT96] of each vertex (where $k = (\log^{(2)} n)^2$). Contracting the edges in these k-Min trees will produce a graph with $O(n/k)$ vertices.

To understand what a k-Min tree is, consider the Dijkstra-Jarnik-Prim minimum spanning tree algorithm:

```
**Dijkstra-Jarnik-Prim**($G$)
    $S := \{v\}$ *(choose an arbitrary starting vertex v)*
    $T := \emptyset$
    Repeat until $T$ contains the MST of $G$
        Choose minimum weight edge $(a, b)$ s.t $a \in S$, $b \notin S$
        $T := T + (a, b)$
        $S := S + b$
```

The edge set k-Min(v) consists of the first k edges chosen by this algorithm, when started at vertex v. A forest F is a *k-Min forest* of G if $F \subseteq \mathrm{MSF}(G)$ and for all $v \in G$, $k\text{-Min}(v) \subseteq F$.

Let $P_T(x, y)$ be the set of edges on the path from x to y in tree T, and let $maxweight\{A\}$ be the maximum weight in a set of edges A.

For any forest F in G, define an edge (a, b) in G to be *F-heavy* if $weight(a, b) > maxweight\{P_F(a, b)\}$ and to be *F-light* otherwise. If a and b are not in the same tree in F then (a, b) is F-light.

Let M be the k-Min tree of v. We define $weight_v(w)$ to be $maxweight\{P_M(v, w)\}$ if w appears in k-Min(v), otherwise $weight_v(w) = maxweight\{k\text{-Min}(v)\}$. Define an edge (a, b) to be *k-Min-heavy* if $weight(a, b) > \max\{\mathrm{weight}_a(b), \mathrm{weight}_b(a)\}$, and to be *k-Min-light* otherwise.

Claim 3.1 *Let the measure $weight_v(w)$ be defined with respect to any k in the range [1..n]. Then $weight_v(w) \le maxweight\{P_{MSF}(v, w)\}$.*

Proof: There are two cases, when w falls inside the k-Min tree of v, and when it falls outside. If w is inside k-Min(v), then $weight_v(w)$ is the same as $maxweight\{P_{MSF}(v, w)\}$ since $k\text{-Min}(v) \subseteq MSF$. Now suppose that w falls outside k-Min(v) and $weight_v(w) > maxweight\{P_{MSF}(v, w)\}$. There must be a path from v to w in the MSF consisting of edges lighter than $maxweight\{k\text{-Min}(v)\}$. However, at each step in the Dijkstra-Jarnik-Prim algorithm, at least one edge in P_{MSF} is eligible to be chosen in that step. Since $w \notin k\text{-Min}(v)$, the edge with weight $maxweight\{k\text{-Min}(v)\}$ is never chosen. Contradiction. □

Let K be a vector of n values, each in the range $[1..n]$. Each vertex u is associated with a value of K, denoted k_u. Define an edge (u, v) to be K-Min-light if $weight(u, v) < \max\{weight_u(v), weight_v(u)\}$, where $weight_u(v)$ and $weight_v(u)$ are defined with respect to k_u and k_v respectively.

Lemma 3.1 *Let H be a graph formed by sampling each edge in graph G with probability p. The expected number of edges in G that are K-Min-light in H is less than n/p, for any K.*
Proof: We show that any edge that is K-Min-light in G is also F-light where F is the MSF of H. The lemma then follows from the sampling lemma of [KKT95] which states that the expected number of F-light edges in G is less than n/p. Let us look at any K-Min-light edge (v, w). By Claim 3.1, $weight_v(w) \leq maxweight\{P_{MSF}(v, w)\}$, the measure used to determine F-lightness. Thus the criterion for K-Min-lightness, $\max\{weight_v(w), weight_w(v)\}$, must also be less than or equal to $maxweight\{P_{MSF}(v, w)\}$. Restating this, if (v, w) is K-Min-light, it must be F-light as well. $\square$

We will use the above property of a k-Min forest to develop a procedure Find-k-Min(G, l). It takes as input the graph G and a suitable positive integer l, and returns a k-Min forest of G. For $l = \log^* n$, it runs in logarithmic time and linear work. In the next few sections we describe some basic steps and procedures used in Find-k-Min, and then present and analyze this main procedure of Phase 1.

Since Phase 1 is concerned only with the k-Min tree of each vertex, it suffices to retain only the lightest k edges incident on each vertex. Hence as stated in the first step of Phase 1 in algorithm High-Level in Section 2 we will discard all but the lightest k edges incident on each vertex since we will not need them until Phase 2. This step can be performed in logarithmic time and linear work by a simple randomized algorithm that selects a sample of size $\sqrt{|L|}$ from each adjacency list L, sorts this sample, and then uses this sorted list to narrow the search for the kth smallest element to a list of size $O(|L|^{3/4})$.

3.2 Borůvka-A Steps

In a basic Borůvka step [Bor26], each vertex chooses its minimum weight incident edge, inducing a number of disjoint trees. All such trees are then contracted into single vertices, and useless edges discarded. We will call edges connecting two vertices in the same tree *internal* and all others *external*. All internal edges are useless, and if multiple external edges join the same two trees, all but the lightest are useless.

Our algorithm for Phase 1 uses a modified Borůvka step in order to reduce the time bound to $o(\log n)$ per step. All vertices are classified as being either *live* or *dead*. After a modified Borůvka step, vertex v's *parent pointer* is $p(v) = w$, where (v, w) is the edge of minimum weight incident on v. In addition, each vertex has a *threshold* which keeps the weight of the lightest discarded edge adjacent to v. The algorithm discards edges known not to be in the k-Min tree of any vertex. The threshold variable guards against vertices choosing edges which may not be in the MSF. A dead vertex v has the useful property (shown below) that for any edge (a, b) in k-Min(v), weight$(a, b) \leq$ weight$(v, p(v))$, thus dead vertices *need not participate* in any more Borůvka steps.

It is well-known that a Borůvka step generates a forest of *pseudo-trees*, where each pseudo-tree is a tree together with one extra edge that forms a cycle of length 2. In our algorithm we will assume that a Borůvka step also removes one of the edges in the cycle so that it generates a collection of rooted trees.

The following three claims refer to any tree resulting from a modified Borůvka step. Their proofs are straightforward and are omitted.

Claim 3.2 *The sequence of edge weights encountered on a path from v to root(v) is monotonically decreasing.*

Claim 3.3 *If depth(v) $= d$ then d-Min(v) consists of the edges in the path from v to root(v). Furthermore, the weight of $(v, p(v))$ is greater than any other edge in d-Min(v).*

Claim 3.4 *If the minimum-weight incident edge of u is (u,v), k-Min$(u) \subseteq (k\text{-Min}(v) + (u,v))$.*

Claim 3.5 *Let T be a tree induced by a Borůvka step, and let T' be a subtree of T. If e is the minimum weight incident edge on T, then the minimum weight incident edge on T' is either e or an edge of T.*

Proof: Suppose, on the contrary that the minimum weight incident edge on T' is $e' \notin T$, and let v and v' be the end points of e and e' which are inside T. Consider the paths P (P') from v (v') to the root of T. By Claim 3.2, the edge weights encountered on P and P' are monotonically decreasing. There are two cases. If T' contains some, but not all of P', then e' must lie along P'. Contradiction. If T' contains all of P', but only some of P, then some edge $e'' \in P$ is adjacent to T'. Then $w(e') < w(e'') < w(e)$, also a contradiction. □

The procedure Borůvka-A(H,l,F) given below returns a contracted version of H with the number of live vertices reduced by a factor of l. Edges designated as parent pointers, which are guaranteed to be in the MSF of H, are returned in F. Initially $F = \emptyset$.

```
Borůvka-A(H, l, F)
        Repeat $\log l$ times: ($\log l$ modified Borůvka steps)
                $F' := \emptyset$
                For each live vertex $v$
                        Choose min. weight edge $(v,w)$
(1)                     If weight$(v,w) >$ threshold$(v)$, $v$ becomes dead, stop else
                        p$(v) := w$
                        $F' := F' + (v, \mathrm{p}(v))$
                Each tree $T$ induced by edges of $F'$ is one of two types:
                        If root of $T$ is dead, then
(2)                             Every vertex in $T$ becomes dead (Claim 3.4)
                        If $T$ contains only live vertices
(3)                             If depth$(v) \geq k$, $v$ becomes dead (Claim 3.3)
                                Contract the subtree of $T$ made up of live vertices
                                The resulting vertex is live, has no parent pointer, and
                                keeps the smallest threshold of its constituent vertices
                $F := F + F'$
```

Lemma 3.2 *If Borůvka-A designates a vertex as dead, its k-Min tree has already been found.*

Proof: Vertices make the transition from live to dead only at the lines indicated by a number. By our assumption that we only discard edges that cannot be in the k-Min tree of any vertex, if the lightest edge adjacent to any vertex has been discarded, we know its k-Min tree has already been found. This covers line (1). The correctness of line (2) follows from Claim 3.4. Since $(v, p(v))$ is the lightest incident edge on v, k-Min$(v) \subseteq k$-Min$(p(v)) + (v, p(v))$. If $p(v)$ is dead, then v can also be called dead. Since the root of a tree is dead, vertices at depth one are dead, implying vertices at depth two are dead, and so on. The validity of line (3) follows directly from Claim 3.3. If a vertex finds itself at depth $\geq k$, its k-Min tree lies along the path from the vertex to its root. □

Lemma 3.3 *After a call to Borůvka-A$(H, k+1, F)$, the k-Min tree of each vertex is a subset of F.*

Proof: By Lemma 3.2, dead vertices already satisfy the lemma. After a single modified Borůvka step, the set of parent pointers associated with live vertices induce a number of trees. Let $T(v)$ be the tree containing v. We assume inductively that after $\lceil \log i \rceil$ modified Borůvka steps, the $(i-1)$-Min tree of each vertex in the original graph has been found (this is clearly true for $i = 1$). For any live vertex v let (x, y) be the minimum weight edge s.t. $x \in T(v)$, $y \notin T(v)$. By the inductive hypothesis, the $(i-1)$-Min trees of v and y are subsets of $T(v)$ and $T(y)$ respectively. By Claim 3.5, (x, y) is the first external edge of $T(v)$ chosen by the Dijkstra-Jarnik-Prim algorithm, starting at v. As every edge in $(i-1)$-Min(y) is lighter than (x, y), $(2(i-1)+1)$-Min(v) is a subset of $T(v) \cup \{(x, y)\} \cup T(y)$. Since edge (x, y) is chosen in the $(\lceil \log i \rceil + 1)^{th}$ modified Borůvka step, $(2i-1)$-Min(v) is a subset of $T(v)$ after $\lceil \log i \rceil + 1 = \lceil \log 2i \rceil$ modified Borůvka steps. Thus after $\log(k+1)$ steps, the k-Min tree of each vertex has been found. □

Lemma 3.4 *After b modified Borůvka steps, the length of any edge list is bounded by k^{k^b}.*

Proof: This is true for $b = 0$. Assuming the lemma holds for $b-1$ modified Borůvka steps, the length of any edge list after that many steps is $\leq k^{k^{b-1}}$. Since we only contract trees of height $< k$, the length of any edge list after b steps is $< (k^{k^{b-1}})^k = k^{k^b}$. □

It is shown in the next section that our algorithm only deals with graphs that are the result of $O(\log k)$ modified Borůvka steps. Hence the maximum length edge list is $k^{k^{O(\log k)}}$.

The costliest step in Borůvka-A is calculating the depth of each vertex. After the minimum weight edge selection process, the root of each induced tree will broadcast its depth to all depth 1 vertices, which in turn broadcast to depth 2 vertices, etc. Once a vertex knows it is at depth $k-1$, it may stop, letting all its descendents infer that they are at depth $\geq k$. Interleaved with each round of broadcasting is a processor allocation step. We account for this cost separately in section 7.

Lemma 3.5 *Let G_1 have m_1 edges. Then a call to Borůvka-A(G_1, l, F) can be executed in time $O(k^{O(\log k)} + \log l \cdot \log n \cdot (m_1/m))$ with $(m+n)/\log n$ processors.*

Proof: Let G_1 be the result of b modified Borůvka steps. By Lemma 3.4, the maximum degree of any vertex after the i^{th} modified Borůvka step in the current call to Borůvka-A is $k^{k^{b+i}}$. Let us now look at the required time of the i^{th} modified Borůvka step. Selecting the minimum cost incident edge takes time $\log k^{k^{b+i}}$, while the time to determine the depth of each vertex is $k \cdot \log k^{k^{b+i}}$. Summing over the $\log l$ modified Borůvka steps, the total time is bounded by $\sum_i^{\log l} k^{O(b+i)} = k^{O(b+\log l)}$. As noted above, the algorithm performs $O(\log k)$ modified Borůvka steps on any graph, hence the time is $k^{O(\log k)}$.

The work performed in each modified Borůvka step is linear in the number of edges. Summing over $\log l$ such steps and dividing by the number of processors, we arrive at the second term in the stated running time. □

3.3 The Filtering Step

The Filter Forest

Concurrent with each modified Borůvka step, we will maintain a Filter forest, a structure that records which vertices merged together at what time, and the edge weights involved. (This structure appeared first in [King97]). If v is a vertex of the original graph, or a new vertex resulting from contracting a set of edges, there is a corresponding vertex $\phi(v)$ in the Filter forest. During a Borůvka step, if a vertex v becomes dead, a new vertex w is added to the Filter forest, as well as a directed edge $(\phi(v), w)$ having the same weight as $(v, p(v))$. If live vertices $v_1, v_2, \ldots, v_j$ are contracted into a live vertex v, a vertex $\phi(v)$ is added to the Filter forest in addition to directed edges $(\phi(v_1), \phi(v)), (\phi(v_2), \phi(v)), \ldots, (\phi(v_j), \phi(v))$, having the weights of edges $(v_1,\mathrm{p}(v_1))$, $(v_2,\mathrm{p}(v_2)), \ldots,$ $(v_j,\mathrm{p}(v_j))$, respectively.

The measures $\mathrm{weight}_v(w)$ can be easily computed in the following way. Let $P_f(x, y)$ be the path from x to y in the Filter forest. If $\phi(v)$ and $\phi(w)$ are not in the same Filter tree, then

$$\mathrm{weight}_v(w) = maxweight\{P_f(\phi(v), \mathrm{root}(\phi(v)))\} \text{ and}$$
$$\mathrm{weight}_w(v) = maxweight\{P_f(\phi(w), \mathrm{root}(\phi(w)))\}$$

If v and w are in the same Filter tree, let $\mathrm{LCA} = \mathrm{LCA}(\phi(v), \phi(w))$, then

$$\mathrm{weight}_v(w) = \mathrm{weight}_w(v) = \max\{maxweight\{P_f(\phi(v), \mathrm{LCA})\}, maxweight\{P_f(\phi(w), \mathrm{LCA})\}$$

It is shown in [King97] that the heaviest weight in the path from u to v in the MSF is the same as the heaviest weight in the path from $\phi(u)$ to $\phi(v)$ in the Filter forest (if there is such a path).

Claim 3.6 *The maximum weight on the path from $\phi(v)$ to $root(\phi(v))$ is the same as the maximum weight edge in r-Min(v), for some r.*

Proof: If $root(\phi(v))$ is at height h, then it is the result of h Borůvka steps. Assume that the claim holds for the first $i < h$ Borůvka steps. After a number of contractions, vertex v of the original graph is now represented in the current graph by v_c. Let T_{v_c} be the tree induced by the i^{th} Borůvka step which contains v_c, and let e be the minimum weight incident edge on T_{v_c}. By the inductive hypothesis, $maxweight\{P_f(\phi(v), \phi(T_{v_c}))\} = maxweight\{r'\text{-Min}(v)\}$ for some r'. As was shown in the proof of Claim 3.5, all edges on the path from v_c to edge e have weight at most $\max\{weight(v_c, p(v_c)), weight(e)\}$. Each of the edges $(v_c, p(v_c))$ and e has a corresponding edge in the Filter forest, namely $(\phi(v_c), p(\phi(v_c)))$ and $(\phi(T_{v_c}), p(\phi(T_{v_c})))$. Since both these edges are on the path from $\phi(v)$ to $p(\phi(T_{v_c}))$, $maxweight\{P_f(\phi(v),\, p(\phi(T_{v_c})))\} = maxweight\{r\text{-Min}(v)\}$ for some $r \geq r'$. Thus the claim holds after $i + 1$ Borůvka steps. □

The Filter Step

In a call to Filter(H, F) in Find-k-Min, we examine each edge $e = (x, y)$ in $H - F$, and delete e from H if $\mathrm{weight}(e) > \max\{\mathrm{weight}_v(w),\ \mathrm{weight}_w(v)\}$ In order to carry out this test we can use the $O(\log n)$ time, $O(m)$ work MSF verification algorithm of [KPRS97], where we modify the algorithm for the case when x and y are not in the same tree to test the pairs $(\phi(x), root(\phi(x)))$ and $(\phi(y), root(\phi(y)))$, and we delete e if both of these pairs are identified to be deleted. This computation will take time $O(\log r)$ where r is the size of the largest tree formed.

The procedure Filter discards edges that cannot be in the k-Min tree of any vertex. When it discards an edge (a, b), it updates the *threshold* variables of both a and b, so that threshold(a) is the weight of the lightest discarded edge adjacent to a. If a's minimum weight edge is ever heavier than threshold(a), k-Min(a) has already been found, and a becomes dead.

Claim 3.7 *Let H' be a graph formed by sampling each edge in H with probability p, and F be a k-Min forest of H'. The call to Filter(H, F) returns a graph containing a k-Min forest of H, whose expected number of edges is n/p.*

Proof: For each vertex v, Claim 3.6 states that $maxweight\{P_f(\phi(v), root(\phi(v)))\} = maxweight\{k_v\text{-}$ Min(v) for *some* value k_v. By building a vector K of such values, one for each vertex, we are able to check for K-Min-lightness using the Filter forest. It follows from Lemma 3.1 that the expected number of K-Min-light edges in H is less than n/p. Now we need only show that a k-Min-light edge of H is not removed in the Filter step. Suppose that edge (u, v) is in the k-Min tree of u in H, but is removed by Filter. If v is in the k_u-Min tree of u (w.r.t. H'), then edge (u, v) was the heaviest edge in a cycle and could not have been in the MSF, much less any k-Min tree. If v was not in the k_u-Min tree of u (w.r.t. H'), then $weight(u, v) > maxweight\{k_u\text{-Min}(u)\}$, meaning edge (u, v) could not have been picked in the first k steps of the Dijkstra-Jarnik-Prim algorithm. □

3.4 Finding a k-Min Forest

We are now ready to present the main procedure of Phase 1, Find-k-Min. (Recall that the initial call – given in Section 2 – is Find-k-Min($G_t, \log^* n$), where G_t is the graph obtained from G by removing all but the k lightest edges on each adjacency list.)

```
Find-k-Min($H, i$)
    $H_c$ := Borůvka-A($H, (\log^{(i-1)} n)^4, F$)
    if $i = 3$, return($F$)
    $H_s$ := sample edges of $H_c$ with prob. $1/(\log^{(i-1)} n)^2$
    $F_s$ := Find-k-Min($H_s, i-1$)
    $H_f$ := Filter($H_c, F_s$)
    $F'$ := Find-k-Min($H_f, i-1$)
    Return($F + F'$)
```

H is a graph with some vertices possibly marked as dead; i is a parameter that indicates the level of recursion (which determines the number of Borůvka steps to be performed and the sampling probability).

Lemma 3.6 *The call Find-k-Min($G_t, \log^* n$) returns a set of edges that includes the k-Min tree of each vertex in G_t.*

Proof: The proof is by induction on i.
Base: $i = 3$. Then Find-k-Min($H, 3$) returns F, which by Lemma 3.3 contains the k-min tree of each vertex.
Induction Step: Assume inductively that Find-k-Min($H, i-1$) returns the k-min tree of H. Consider the call Find-k-Min(H, i). By the induction assumption the call to Find-k-Min($H_s, i-1$) returns the k-min tree of each vertex in H_s. By Claim 3.7 the call to Filter(H_c, F_s) returns in H_f a set of edges that contains the k-Min trees of all vertices in H_c. Finally, by the inductive assumption, the set of edges returned by the call to Find-k-min($H_f, i-1$) contains the k-Min trees of all vertices in H_f. Since F' contains the ($\log^{(i-1)} n$)-Min tree of each vertex in H, and Find-k-Min(H, i) returns $F + F'$, it returns the edges in the k-Min tree of each vertex in H. □

Claim 3.8 *The following invariants are maintained at each call to Find-k-min. The number of live vertices in $H \leq n/(\log^{(i)} n)^4$, and the expected number of edges in $H \leq m/(\log^{(i)} n)^2$, where m and n are the number of edges and vertices in the original graph.*

Proof: These clearly hold for the initial call, when $i = \log^* n$. By Lemma 3.3, the contracted graph H_c has no more than $n/(\log^{(i-1)} n)^4$ live vertices. Since H_s is derived by sampling edges with probability $1/(\log^{(i-1)} n)^2$, the expected number of edges in H_s is $\leq m/(\log^{(i-1)} n)^2$, maintaining the invariants for the first recursive call.

By Lemma 3.1, the expected number of edges in $H_f \le \frac{n(\log^{(i-1)} n)^2}{(\log^{(i-1)} n)^4} = n/(\log^{(i-1)} n)^2$. Since H_f has the same number of vertices as H_c, both invariants are maintained for the second recursive call. □

3.5 Performance of Find-k-Min

Lemma 3.7 *Find-k-min($G_t, \log^* n$) runs in expected time $O(\log n)$ and work $O(m+n)$.*

Proof: Since recursive calls to Find-k-min proceed in a sequential fashion, the total running time is the sum of the local computation performed in each invocation. Aside from randomly sampling the edges, which takes constant time and work linear in the number of edges, the local computation consists of calls to Filter and Borůvka-A.

In a given invocation of Find-k-min, the number of Borůvka steps performed on graph H is the sum of all Borůvka steps performed in all ancestral invocations of Find-k-min, i.e. $\sum_{i=3}^{\log^* n} O(\log^{(i)} n)$, which is $O(\log^{(3)} n)$. ¿From our bound on the maximum length of edge lists (Lemma 3.4), we can infer that the size of any tree in the Filter forest is $k^{k^{O(\log^{(3)} n)}}$, thus the time needed for each modified Borůvka step and each Filter step is $k^{O(\log^{(3)} n)}$. Summing over all such steps, the total time required is $o(\log n)$.

The work required by the Filter procedure and *each* Borůvka step is linear in the number of edges. As the number of edges in any given invocation is $O(m/(\log^{(i)} n)^2)$, and there are $O(\log^{(i)} n)$ Borůvka steps performed in this invocation, the work required in each invocation is $O(m/\log^{(i)} n)$ (recall that the i parameter indicates the depth of recursion). Since there are $2^{\log^* n - i}$ invocations with depth parameter i, the total work is given by $\sum_{i=3}^{\log^* n} 2^{\log^* n - i} O(m/\log^{(i)} n)$, which is $O(m)$. □

4 Phase 2

Recall the Phase 2 portion of our overall algorithm **High-Level**:

```
(the number of vertices in $G_s$ is $\le n/k$)
$G_s$ :=Sample edges of $G'$ with prob. $1/\sqrt{k} = 1/\log^{(2)} n$
$F_s$ :=Find-MSF($G_s$)
$G_f$ := Filter($G', F_s$)
$F$ := Find-MSF($G_f$)
```

The procedure Filter(G, F) ([KPRS97]) returns the F-light edges of G. The procedure Find-MSF(G_1), described below, finds the MSF of G_1 in time $O(\log n \log^{(2)} n\,(m_1/m))$, where m_1 is the number of edges in G_1. Because we call Find-MSF on graphs having expected size $O(m/\log^{(2)} n)$, each call takes $O(\log n)$ time.

The graphs G_s and G_f each have expected $m/\sqrt{k} = m/\log^{(2)} n$ edges. G_s is derived by sampling each edge with prob $1/\sqrt{k}$, and by the sampling lemma of [KKT95], the expected number of edges in G_f is $(m/k)/(1/\sqrt{k}) = m/\sqrt{k}$.

4.1 The Find-MSF Procedure

The procedure Find-MSF(H) is similar to previous randomized parallel algorithms, except it uses no recursion. Instead, a separate *base case* algorithm is used in place of recursive calls. We also use slightly different Borůvka steps, in order to reduce the work. These modifications are inspired by [PR97] and [PR98] respectively.

As its Base-case, we use the algorithm of Chong, Han, and Lam [CHL99], which takes time $O(\log n)$ using $m+n$ processors. By guaranteeing that it is only called on graphs of expected size $O(m/\log n)$, the running time remains $O(\log n)$ with $(m+n)/\log n$ processors.

```
Find-MSF($H$)
  $H_c$ = Borůvka-B($H, \log^2 n, F$)
  $H_s$ = Sample edges of $H_c$ with prob. $p = 1/\log n$
  $F_s$ = BaseCase($H_s$)
  $H_f$ = Filter($H_c, F_s$)
  $F'$ = BaseCase($H_f$)
  Return($F + F'$)
```

After the call to Borůvka-B, the graph H_c has $< n/\log^2 n$ vertices. Since H_s is derived by sampling the edges of H_c with probability $1/\log n$, the expected number of edges to the first BaseCase call is $O(m/\log n)$. By the sampling lemma of [KKT95], the expected number of edges to the second BaseCase call is $< (n/\log^2 n)/(1/\log n)$, thus the total time spent in these subcalls is $O(\log n)$. Assuming the size of H conforms to its expectation of $O(m/\log^{(2)} n)$, the calls to Filter and Borůvka-B also take $O(\log n)$ time, as described below.

The Borůvka-B(H, l, F) procedure returns a contracted version of H with $\leq m/l$ vertices. It uses a simple growth control schedule, designating vertices as *inactive* if their degree exceeds l. We can determine if a vertex is inactive by performing list ranking on its edge list for $\log l$ time steps. If the computation has not stopped after this much time, then its edge list has length $> l$.

```
Borůvka-B($G, l, F$)
  Repeat $\log l$ times
      For each vertex, let it be inactive if its edge list
      has more than $l$ edges, and active otherwise.
      For each active vertex $v$
          choose min. weight incident edge $e$
          $F = F + e$
      Using the edge-plugging technique, build a
      single edge list for each induced tree ($O(1)$ time)
  Contract all trees of inactive vertices
```

The last step takes $O(\log n)$ time; all other steps take $O(\log l)$ time, as they deal with edge lists of length $O(l)$. Consequently, the total running time is $O(\log n + \log^2 l)$. For each iteration of the main loop, the work is linear in the number of edges. Assuming the graph conforms to its expected size of $O(m/\log^{(2)} n)$, the total work is linear. The edge-plugging technique was first used by Johnson & Metaxas [JM92], as well as the idea of a growth control schedule.

5 Proof of Main Theorem

Proof: (of Theorem 2.1) The set of edges M returned by Find-k-Min is a subset of the MSF of G. By contracting the edges of M to produce G', the MSF of G is given by the edges of M together with the MSF of G'. The call to Filter produces graph G_f by removing from G' edges known not to be in the MSF. Thus the MSF of G_f is the same as the MSF of G'. Assuming the correctness of Find-MSF, the set of edges F constitutes the MSF of G_f, thus $M + F$ is the MSF of G.

Earlier we have shown that each step of High-Level requires $O(\log n)$ time and work linear in the number of edges. In the next two sections we show that w.h.p, the number of edges encountered in all graphs during the algorithm is linear in the size of the original graph. $\Box$

6 High Probability Bounds

Consider a single invocation of Find-k-min(H, i), where H has m' edges and n' vertices. We want to place likely bounds on the number of edges in each recursive call to Find-k-min, in terms of m' and i.

For the first recursive call, the edges of H are sampled independently with probability $1/(\log^{(i-1)} n)^2$. Call the sampled graph H_1. By applying a Chernoff bound, the probability that the size of H_1 is less than twice its expectation is $1 - \exp(-\Omega(m'/(\log^{(i-1)} n)^2))$.

Before analyzing the second recursive call, we recall the sampling lemma of [KKT95] which states that the number of F-light edges conforms to the negative binomial distribution with parameters n' and p, where p is the sampling probability, and F is the MSF of H_1. As we saw in the proof of Lemma 3.1, every k-Min-light edge must also be F-light. Using this observation, we will analyze the size of the second recursive call in terms of F-light edges, and conclude that any bounds we attain apply equally to k-Min-light edges.

We now bound the likelihood that more than twice the expected number of edges are F-light. This is the probability that in a sequence of more than $2n'/p$ flips of a coin, with probability p of heads, the coin comes up heads less than n' times (since each edge selected by a coin toss of heads goes into the MSF of the sampled graph). By applying a Chernoff bound, this is $\exp(-\Omega(n'))$. In this particular instance of Find-k-min, $n' \leq m/(\log^{(i-1)} n)^4$ and $p = 1/(\log^{(i-1)} n)^2$, so the probability that fewer than $2m/(\log^{(i-1)} n)^2$ edges are F-light is $1 - \exp(-\Omega(m/(\log^{(i-1)} n)^4))$.

Given a single invocation of Find-k-min(H, i), we can bound the probability that H has more than $2^{\log^* n - i} m/(\log^{(i)} n)^2$ edges by $\exp(-\Omega(m/(\log^{(i)} n)^4))$. This follows from applying the argument used above to each invocation of Find-k-min from the initial call to the current call at depth $\log^* n - i$. Summing over all recursive calls to Find-k-min, the total number of edges (and thus the total work) is bounded by $\sum_{i=3}^{\log^* n} 2^{2 \log^* n - 2i} m/(\log^{(i)} n)^2 = O(m)$ with probability $1 - \exp(-\Omega(m/(\log^{(3)} n)^4))$.

The analysis of Phase 2 is entirely analogous and much simpler as it does not have to address the effect of recursive calls. We omit the details.

The probability that our bounds on the time and total work performed by the algorithm fail to hold is exponentially small in the input size. However, this assumes perfect processor allocation. In the next section we show that the probability that work fails to be distributed evenly among the processors is less than $1/m^{\omega(1)}$. Thus the overall probability of failure is very small, and the algorithm runs in logarithmic time and linear work w.h.p.

7 Processor Allocation

As stated in Section 2, the processor allocation needed for our algorithm can be performed by a fairly simple algorithm given in [HZ94] that takes logarithmic time and linear work but uses super-linear space, or by a more involved algorithm claimed in [HZ96] that runs in logarithmic time and linear work and space. We show here that a simple algorithm similar in spirit to the one in [HZ94] runs in logarithmic time and linear work and space on the *QRQW PRAM* [GMR94]. The QRQW PRAM is intermediate in power between the EREW and CRCW PRAM in that it allows concurrent memory accesses, but the time taken by such accesses is equal to the largest number of processors accessing any single memory location.

We assume that the total size of our input is n, and that we have $q = n/\log n$ processors. We group the q processors into q/r groups of size $r = \log n$ and we make an initial assignment of $O(r \log n)$ elements to each group. This initial assignment is made by having each element choose a group randomly. The expected number of elements in each group is $r \log n$ and by a Chernoff bound, w.h.p. there are $O(r \log n)$ elements in each group. Vertices assigned to each group can be collected together in an array for that group in $O(\log n)$ time and $O(n)$ work and space by using the QRQW PRAM algorithm for *multiple compaction* given in [GMR96], which runs in logarithmic time and linear work with high probability. (We do not need the full power of the algorithm in [GMR96] since we know ahead of time that each group has $\geq c \log^2 n$ elements w.h.p., for a suitable constant c. Hence it suffices to use the *heavy multiple compaction algorithm* in [GMR96] to achieve the bounds of logarithmic time and linear work and space.)

A simple analysis using Chernoff bounds shows that on each new graph encountered during the computation each group receives either $< \log n$ elements, or within a constant factor of its expected number of elements w.h.p. Hence in $O(\log \log n)$ EREW PRAM steps each processor within a group can be assigned $1/(\log n)$ of the elements in its group. This processor re-allocation scheme takes $O(\log \log n)$ time per stage and linear space overall, and with high probability, achieves perfect balance to within a constant factor. The total number of processor re-allocation steps needed by our algorithm is $O(2^{\log^* n} \cdot k \log k) = O(\log n / \log \log n)$, hence the time needed to perform all of the processor allocation steps is $O(\log n)$ w.h.p.

We note that the probability that processors are allocated optimally (to within a constant factor) can be increased to $1 - n^{-\omega(1)}$ by increasing the group size r. Since we perform $o((\log^{(2)} n)^3)$ processor allocation steps, r can be set as high as $n^{1/(\log^{(2)} n)^3}$ without increasing the overall $O(\log n)$ running time. Thus the high probability bound on the number of items in each group being $O(r \log n)$ becomes $1 - n^{-\omega(1)}$. It is shown in [GMR96] that the heavy multiple compaction algorithm runs in time $O(\log^* n \log m / \log \log m)$ time w.h.p. in m, for any $m > 0$. By choosing $m = n^{\log \log n / \log^* n}$, we obtain $O(\log n)$ running time for this initial step with probability $1 - n^{-\omega(1)}$, which is also the overall probability bound for processor allocation.

8 Adaptations to other Practical Parallel Models

Our results imply good MSF algorithms for the QSM [GMR97] and BSP [Val90] models, which are more realistic models of parallel computation than the PRAM models. Theorem 8.1 given below follows directly from results mapping EREW and QRQW computations on to QSM given in [GMR97]. Theorem 8.2 follows from the QSM to BSP emulation given in [GMR97] in conjunction with the observation that the slowdown in that emulation due to hashing does not occur for our algorithm since the assignment of vertices and edges to processors made by our processor allocation scheme achieves the same effect.

Theorem 8.1 *An MSF of an edge-weighted graph on n nodes and m edges can be found in $O(g \log n)$ time and $O(g(m+n))$ work w.h.p, using $O(m+n)$ space on the QSM with a simple processor allocation scheme, where g is the gap parameter of the QSM.*

Theorem 8.2 *An MSF of an edge-weighted graph on n nodes and m edges can be found on the BSP in $O((L+g)\log n)$ time w.h.p., using $(m+n)/\log n$ processors and $O(m+n)$ space with a simple processor allocation scheme, where g and L are the gap and periodicity parameters of the BSP.*

References

[AS87] B. Awerbuch, Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange networks and PRAM. *IEEE Trans. Computers*, vol. C-36, 1987, pp. 1258-1263.

[Bor26] O. Borůvka . O jistém problému minimaálním. *Moravské Přírodovědecké Společnosti* 3, (1926), pp. 37-58. (In Czech).

[CHL99] K. W. Chong, Y. Han and T. W. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. In *Proc. SODA* 1999.

[CKT94] R. Cole, P.N. Klein, and R.E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. In *Proc. SPAA*, 1994, pp. 11–15.

[CKT96] R. Cole, P.N. Klein, and R.E. Tarjan. Finding minimum spanning trees in logarithmic time and linear work using random sampling. In *Proc. SPAA*, 1996, pp. 213–219.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. In *Numer. Math.*, 1 (1959), pp. 269-271.

[Gaz91] H. Gazit An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, vol. 20, 1991, pp. 1046-1067.

[GMR94] P.B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. SODA*, 1994, pp. 638–648.

[GMR96] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *Jour. Comput. Systems Sciences*, vol. 53, 1996, pp. 395-416.

[GMR97] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. SPAA*, 1997, pp. 72–83.

[HZ94] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. In *Proc. SPAA*, 1994, pp. 1-10.

[HZ96] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *Proc. SODA*, 1996, pp. 438–447, 1996.

[Jar30] V. Jarník. O jistém problému minimaálním. *Moravské Přírodovědecké Společnosti* 6, 1930, pp. 57-63. (In Czech).

[JM92] D. B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for CREW PRAM. *Jour. Comput. Sys. Sciences*, vol. 54, 1997, pp. 227–242.

[King97] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, vol. 18, 1997, pp. 263-270.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[KPRS97] V. King, C. K. Poon, V. Ramachandran, and S. Sinha. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Information Processing Letters*, 62(3):153–159, 1997.

[KR90] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, Vol. A, 1990, pp. 869-941. Elsevier Science, The Netherlands.

[PR97] C. K. Poon, V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. *Proc ISAAC*, 1997, pp. 212.-222.

[PR98] C. K. Poon, V. Ramachandran. Private communication, 1998.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Prim57] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389-1401.