SIAM J. COMPUT.
Vol. 31, No. 6, pp. 1879–1895

© 2002 Society for Industrial and Applied Mathematics

A RANDOMIZED TIME-WORK OPTIMAL PARALLEL ALGORITHM FOR FINDING A MINIMUM SPANNING FOREST*

SETH PETTIE† AND VIJAYA RAMACHANDRAN†

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 exclusive read exclusive write (EREW) PRAM. This result is optimal w.r.t. both work and parallel time, and is the first provably optimal parallel algorithm for this problem under both measures. We also give a simple, general processor allocation scheme for tree-like computations.

Key words. parallel algorithm, minimum spanning tree, optimal algorithm, EREW PRAM

AMS subject classifications. 05C85, 68R10, 68Q85

PII. S0097539700371065

1. Introduction. We present a randomized parallel algorithm to find a minimum spanning forest (MSF) in an edge-weighted, undirected graph. On an exclusive read exclusive write (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 w.r.t. 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 MSF algorithms for the concurrent read concurrent write (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 Chong, Han, and Lam [CHL01] presented a deterministic EREW PRAM algorithm for MSF, 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 sped up to run in $O(\log n \cdot 2^{\log^* n})$ time and linear work by using the algorithm in [CHL01] 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, and we improve on the algorithm in [CKT96] by achieving the logarithmic time bound on the less powerful EREW PRAM.

*Received by the editors April 19, 2000; accepted for publication (in revised form) March 20, 2002; published electronically October 18, 2002. This work was supported by Texas Advanced Research Program grant 003658-0029-1999. A preliminary version of this paper appeared in *Randomization, Approximation, and Combinatorial Optimization (Berkeley, CA,* 1999*)*, Lecture Notes in Comput. Sci. 1671, Springer-Verlag, Berlin, 1999, pp. 233–244.

http://www.siam.org/journals/sicomp/31-6/37106.html

†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712 (seth@cs.utexas.edu, vlr@cs.utexas.edu). The research of the first author was supported by an MCD graduate fellowship. The research of the second author was supported by NSF grant CCR-9988160.

Our algorithm has a simple 2-phase structure. It makes subroutine calls to the Chong–Han–Lam (CHL) algorithm [CHL01], which is fairly complex. But outside of these subroutine calls (which are made to the simplest version of the algorithm in [CHL01]), the steps in our algorithm are quite straightforward.

In addition to being the first time-work optimal parallel algorithm for MSF, 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 the (work inefficient) scheme used in [CKT96]. The resulting algorithm runs in logarithmic time and linear work but is considerably simpler than the MSF algorithm in [CKT96] and also is about twice as fast.
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 [HZ96, HZ01] since we compute a *minimum* spanning tree within the same time and work bounds. Our algorithm also is simpler than the algorithms in [HZ96, HZ01].

In the following we use the notation $S + T$ to denote the 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 $c > 0$.

The rest of this paper is organized as follows. In section 2 we give a high-level description of our algorithm, which works in two phases. In section 3 we provide the details of phase 1, whose main purpose is to reduce the number of vertices in the graph by at least a $(\log \log n)^2$ factor. Phase 2, given in section 4, finds the MSF of the reduced-vertex graph using a recursion-free version of the [PR97] algorithm. In sections 5 and 6 we prove the algorithm runs in logarithmic time and linear work with high probability. Section 7 gives a simple processor allocation scheme for "tree-structured" computations (a class containing our MSF algorithm) which is space-optimal. In section 8 we discuss the adaptability of our algorithm to realistic parallel models like the bulk-synchronous parallel (BSP) [Val90] and queuing shared memory models (QSM) [GMR99]. Our conclusions are given in section 9.

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_0 vertices, where n is the number of vertices in the input graph, and[1] $k_0 = (\log^{(2)} n)^2$. To perform this reduction the algorithm uses the familiar recursion tree of depth $\log^* n$ [CKT94, CKT96, PR97], which gives rise to $O(2^{\log^* n})$ recursive calls; however, 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 $O(n/k_0)$ 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 distinct. As always, distinctness can be forced by ordering the vertices, then ordering identically weighted edges by their end points.

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

The high-level algorithm is given in Figure 2.1.

High-Level(G)

(Phase 1)
$G_t :=$ For all $v \in G$, retain the lightest k_0 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. $1/\sqrt{k_0} = 1/\log^{(2)} n$
$F_s :=$ Find-MSF(G_s)
$G_f :=$ Filter(G', F_s)
$F :=$ Find-MSF(G_f)
Return$(M + F)$

FIG. 2.1. *The high-level algorithm.*

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 [HZ96], which requires *superlinear* space. A linear-space load balancing scheme is claimed in [HZ01], though it is difficult to extricate the load balancing computation from the connectivity computation from [HZ01]. In section 7 we give a simple load balancing scheme based on [HZ96], which uses linear space. Its description is abstract and fully self-contained.

3. Phase 1. In phase 1, our goal is to contract the input graph G into a graph with $O(n/k_0)$ vertices. We do this by identifying certain edges in the MSF 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* (for $k = k_0$), 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 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 heavy" edges. For this, 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, called 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. The *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 an MSF, we will construct the *k_0-Min tree* [CKT96] of each vertex (recall that $k_0 = (\log^{(2)} n)^2$). Contracting the edges in these k_0-Min trees will produce a graph with $O(n/k_0)$ vertices.

To understand what a k-Min tree is, consider the Dijkstra–Jarnik–Prim minimum spanning tree algorithm, given in Figure 3.1. This simple algorithm was discovered

independently by Dijkstra [Dij59], Jarnik [Jar30], and Prim [Prim57], though it is commonly known as Prim's algorithm. 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 $F \subseteq \text{MSF}(G)$ and for all $v \in G$, k-Min$(v) \subseteq F$.

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

FIG. 3.1. *The Dijkstra–Jarnik–Prim algorithm.*

We define $P_T(x, y)$ to be the set of edges on the path from x to y in tree T, and $maxweight\{A\}$ 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.

This notion of being F-heavy or F-light can be generalized as follows. Suppose F is a k-Min forest for some graph. We will say an edge (a, b) (not necessarily in the graph) is *k-Min-light* if F is not a k-Min forest for the graph $F+(a, b)$, and *k-Min-heavy* otherwise. An equivalent definition for k-Min-lightness and heaviness (which is more suited to proofs) is this: Let M be the k-Min tree of v. We define $weight_v^k(w)$ to be $maxweight\{P_M(v, w)\}$ if w appears in M, otherwise $weight_v^k(w) = maxweight\{M\}$. An edge (a, b) is then k-Min-light iff $weight(a, b) \leq \max\{weight_a^k(b), weight_b^k(a)\}$. Notice that $weight_v^k(w)$ implicitly depends on the underlying graph, which if not clear from the context will be stated explicitly.

FACT 3.1. *The two definitions given for "k-Min-light" (and "k-Min-heavy") are equivalent.*

We claimed earlier that k-Min-lightness is a generalization of F-lightness. To see this, set $k = n - 1$ and observe that an edge is k-Min-light iff it is F-light.

CLAIM 3.1. *If an edge (u, v) is k_1-Min-heavy w.r.t. G_1, it also is k_2-Min-heavy w.r.t. G_2, where $k_2 \leq k_1$, $V(G_1) = V(G_2)$, and $E(G_1) \subseteq E(G_2)$.*

Proof. This follows from two observations: $weight_v^k(w)$ is nondecreasing in k, and $weight_v^k(w)$ is nonincreasing as new edges are added to the underlyinggraph. □

In other words, whenever an edge is found to be k-Min-heavy for $k \geq k_0$ and w.r.t. some subset of the original graph, this is a certificate that the edge is k_0-Min-heavy in the original graph.

CLAIM 3.2. *For any k, $weight_v^k(w) \leq maxweight\{P_{MSF}(v, w)\}$.*

Proof. There are two cases: when w falls inside the k-Min tree of v and when it falls outside. If w lies inside k-Min(v), then $weight_v^k(w)$ must be the same as $maxweight\{P_{MSF}(v, w)\}$ since k-Min$(v) \subseteq MSF$. Now suppose that w falls outside k-Min(v) and $weight_v^k(w) > maxweight\{P_{MSF}(v, w)\}$. 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 $P_{MSF}(v, w)$ is eligible to be chosen in that step. Since $w \notin k$-Min(v), the edge with weight $maxweight\{k\text{-Min}(v)\}$ is never chosen, a contradiction. □

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 w.r.t. H for any k is less than n/p.*

Proof. We show that any edge that is k-Min-light in G also is 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 3.2, $weight_v^k(w) \leq maxweight\{P_{MSF}(v, w)\}$, the measure used to determine F-lightness. Thus the criterion for k-Min-lightness, $\max\{weight_v^k(w), weight_w^k(v)\}$, must also be no more than $maxweight\{P_{MSF}(v, w)\}$. Restating this, if (v, w) is k-Min-light, it must be F-light as well. □

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_0-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_0-Min tree of each vertex, it suffices to retain only the lightest k_0 edges incident on each vertex. Hence, as stated in the first step of phase 1 in algorithm High-Level from section 2, we will discard all but the lightest k_0 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 k_0th 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 the useless edges are 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*; only live vertices participate in modified Borůvka steps. After such a 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_0-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 e in k_0-Min(v), weight$(e) \leq$ 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 3.3. *The sequence of edge weights encountered on a path from v to root(v) is monotonically decreasing.*

CLAIM 3.4. *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 3.5. *If the minimum-weight incident edge of u is (u,v), k-Min$(u) \subseteq (k$-Min$(v) + (u,v))$.*

Claim 3.6 may not be as obvious. A similar claim was proved in [CHL01].

CLAIM 3.6. *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 3.3, 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', a 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 in Figure 3.2, 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$(v) := w$
$F' := F' + (v,$ 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 3.5)*
If T contains only live vertices,
(3) If depth$(v) \geq k_0$, v becomes dead *(Claim 3.4)*
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.
$F := F + F'$

FIG. 3.2. *The Borůvka-A procedure.*

LEMMA 3.2. *If Borůvka-A designates a vertex as dead, its k_0-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 discard only edges that cannot be in the k_0-Min tree of any vertex, if the lightest edge adjacent to any vertex has been discarded, we know its k_0-Min tree has already been found. This covers line (1). The correctness of line (2) follows from Claim 3.5. Since $(v, p(v))$ is the lightest incident edge on v, k_0-Min$(v) \subseteq k_0$-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

3.4. If a vertex finds itself at depth $\geq k_0$, its k_0-Min tree lies along the path from the vertex to its root. □

LEMMA 3.3. *After a call to Borůvka-A(H, k_0+1, F), the k_0-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 3.6, (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 $(\lceil \log i \rceil + 1)$st 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 $\log(k_0+1)$ steps, the k_0-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 ${k_0}^{{k_0}^b}$.*

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 $\leq {k_0}^{{k_0}^{b-1}}$. Since we contract only trees of height $< k_0$, the length of any edge list after b steps is $< ({k_0}^{{k_0}^{b-1}})^{k_0} = {k_0}^{{k_0}^b}$. □

It is shown in the next section that our algorithm deals only with graphs that are the result of $O(\log k_0)$ modified Borůvka steps. Hence the maximum length edge list is ${k_0}^{{k_0}^{O(\log k_0)}}$.

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 $k_0 - 1$, it may stop, letting all its descendents infer that they are at depth $\geq k_0$. 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 $O({k_0}^{O(\log k_0)} + \log l \cdot \log n \cdot (m_1/m))$ 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 ith modified Borůvka step in the current call to Borůvka-A is ${k_0}^{{k_0}^{b+i}}$. Let us now look at the required time of the ith modified Borůvka step. Selecting the minimum cost incident edge takes time $O(\log {k_0}^{{k_0}^{b+i}})$, while the time to determine the depth of each vertex is $O(k_0 \cdot \log {k_0}^{{k_0}^{b+i}})$. Summing over the $\log l$ modified Borůvka steps, the total time is bounded by $\sum_i^{\log l} {k_0}^{O(b+i)} = {k_0}^{O(b+\log l)}$. As noted above, the algorithm performs $O(\log k_0)$ modified Borůvka steps on any graph, hence the time is ${k_0}^{O(\log k_0)}$.

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. Filtering edges via the Filter forest. We will maintain, concurrent with the operation of Borůvka-A, a structure called the *Filter forest*. This collec-

tion of rooted trees records which vertices merged together and the edge weights involved. (This structure appeared first in [K97].) 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 x is added to the Filter forest, as well as a directed edge $(\phi(v), x)$ 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 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. We make the simple observation that the edge weights on the path from $\phi(u)$ to $root(\phi(u))$ are exactly the edge weights of the edges chosen by u (or its representative) in previous Borůvka steps.

It is shown in [K97] 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). We extend this scheme to handle k-Min-lightness.

Let $P_f(y, z)$ be the path from y to z in the Filter forest. If $\phi(u)$ and $\phi(v)$ are in the same Filter tree, then let

$$w_u(v) = w_v(u) = maxweight\{P_f(\phi(u), \phi(v))\}.$$

If $\phi(u)$ and $\phi(v)$ are not in the same Filter tree, then let

$$w_u(v) = maxweight\{P_f(\phi(u), \mathrm{root}(\phi(u)))\},$$
$$w_v(u) = maxweight\{P_f(\phi(v), \mathrm{root}(\phi(v)))\}.$$

In a call to Filter(H, F) (from the procedure Find-k-Min, section 3.4), we examine each edge $e = (u, v)$ in $H - F$ and remove or *filter* e from H if $weight(e) > \max\{w_u(v),\ w_v(u)\}$. Notice that if $w_u(v) = weight_u^{k_0}(v)$ for all v, then we will filter out edges precisely when they are k_0-Min-heavy. We show below that using $w_u(v)$ in lieu of $weight_u^{k_0}(v)$ causes no problems: we retain all k_0-Min-light edges without retaining too many edges in total.

To implement the Filter procedure we use a slight modification to the $O(\log n)$-time, $O(m)$-work MSF verification algorithm of [KPRS97]. If $e = (u, v)$ is the edge being tested and $\phi(u), \phi(v)$ are not in the same Filter tree, we test the pairs $(\phi(u), root(\phi(u)))$ and $(\phi(v), root(\phi(v)))$ instead and delete e if both of these pairs are identified to be deleted. This computation actually takes time $O(\log r)$ where r is the size of the largest tree formed.

Lemmas 3.6 and 3.7, proved below, establish the correctness of the filtering procedure.

LEMMA 3.6. *Suppose b modified Borůvka steps were applied to a graph; then for any vertex u and some $k \geq \min\{k_0, 2^b - 1\}$,*

$$maxweight\{P_f(\phi(u), root(\phi(u)))\} = maxweight\{k\text{-}Min(v)\}.$$

Before proving this we first prove a necessary technical lemma.

LEMMA 3.7. *Let T be a tree of MSF edges after an arbitrary number of Borůvka steps and let $T' = T \cup \{(v, w)\}$, where (v, w), $v \in T$, $w \notin T$ is the edge chosen by T in the next Borůvka step. For any $u \in T$, the maximum weight edge in $P_{T'}(u, w)$ was chosen by the tree containing u in some Borůvka step.*

Proof. Let T be formed after b Borůvka steps. Suppose, without loss of generality, that the lemma is falsified for the first time after the bth Borůvka step. That is, the

heaviest edge in $P_{T'}(u,w)$, say f, was chosen in the bth step. Let $g \neq f$ be the edge chosen by u or u's representative tree in this step. If f lies between g and the root of T, then by Claim 3.3 it is lighter than g, and similarly, if it lies between vertex v and the root of T, then it is lighter than (v,w). Both cases are contradictions. □

We are now ready to prove Lemma 3.6.

Proof. Let $e(u,b)$ be the maximum weight edge chosen by u's tree in the first b Borůvka steps. Assume inductively that $weight(e(u,b-1)) = maxweight\{k(u,b-1)\text{-Min}(u)\}$, where $k(u,b-1) \geq 2^{b-1}-1$ if u is live, $k(u,\cdot) \geq k_0$ if u is dead, and $k(u,b-1)$-Min(u) is contained in a tree of MSF edges after $b-1$ Borůvka steps. If u is dead, it already satisfies the inductive claim for b Borůvka steps, so assume u is alive. Let (z_1,z_2) be the edge chosen by the tree containing u in the bth Borůvka step and let P be the MSF path connecting $k(u,b-1)$-Min(u) to z_1—see Figure 3.3 for a schematic diagram. We have that $weight(z_1,z_2) > weight(e(z_2,b-1))$, because (z_1,z_2) was not already chosen by z_2 in the first $b-1$ steps, and $maxweight\{e(u,b-1)+P+(z_1,z_2)\} = weight(e(u,b))$. This is true because $weight(e(u,b)) = maxweight\{e(u,b-1),(z_1,z_2)\} > maxweight\{P\}$, where the equality is by definition and the inequality is by Lemma 3.7. Let D be the subgraph

$$D = k(u,b-1)\text{-Min}(u) + P + (z_1,z_2) + k(z_2,b-1)\text{-Min}(z_2)$$

and $k(u,b)$ be the smallest number such that $k(u,b)$-Min$(u) \supseteq D$. It follows that $e(u,b)$ is the heaviest edge in $k(u,b)$-Min(u) because when the Dijkstra–Jarnik–Prim algorithm is started from u, until all edges from D are chosen, there is *some* eligible edge from D weighing no more than the edge $e(u,b)$.

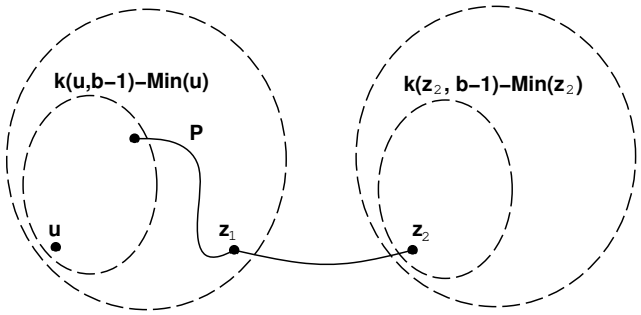

FIG. 3.3. *The larger ovals represent the trees of MSF edges after $b-1$ Borůvka steps containing u and z_2, respectively. The smaller ovals are the $k(u,b-1)$-Min(u) tree and the $k(z_2,b-1)$-Min(z_2) tree.*

If u and z_2 remain live, then $k(u,b) \geq 2\cdot(2^{b-1}-1)+1 \geq 2^b-1$. On the other hand, if u becomes dead after the bth Borůvka step, then (z_1,z_2) is the heaviest edge at the end of a chain C of length at least k_0 and $k(u,b) \geq 2^{b-1}-1+|C| \geq k_0$. In either case our inductive claim is proved for b Borůvka steps. □

LEMMA 3.8. *Suppose the Filter procedure is called only on graphs after performing at least* $\log(k_0+1)$ *Borůvka steps. Then no k_0-Min-light edges are filtered, and all unfiltered edges are k-Min-light for some $k \geq k_0$.*

Proof. Consider an edge (u,v) examined by the Filter procedure. Note that if $\phi(u)$ is in the same Filter tree as $\phi(v)$, by King's observation [K97], $w_u(v) = w_v(u) = maxweight\{P_{MSF}(u,v)\}$. If $weight(u,v)$ is greater than $w_u(v)$, then by Claim 3.1 (u,v) is k_0-Min-heavy and may be safely filtered. On the other hand, if $weight(u,v)$

is less than $w_u(v)$, then (u, v) is k-Min-light for $k = n$. We therefore focus on the case when $\phi(u)$ and $\phi(v)$ are in different Filter trees.

By Lemma 3.6, for some k_1, k_2 we have that $w_u(v) = maxweight\{k_1\text{-Min}(u)\}$ and $w_v(u) = maxweight\{k_2\text{-Min}(v)\}$. Since $maxweight\{k\text{-Min}(u)\}$ is a nondecreasing function of k, if (u, v) is not filtered out, then by Claim 3.1 it must be k_3-Min-light where $k_3 = \max\{k_1, k_2\}$. On the other hand, if (u, v) is filtered out, then it must be k_4-Min-heavy where $k_4 = \min\{k_1, k_2\}$. Because the Filter procedure is applied only after performing at least $\log(k_0 + 1)$ Borůvka steps, by Lemma 3.6 $k_3, k_4 \geq k_0$. □

Remark. Filter is responsible for updating the threshold variables—see section 3.2. When an edge (u, v) is discarded, threshold(u) is updated to reflect the weight of the lightest discarded edge incident to u; threshold(v) is updated similarly.

3.4. Finding a *k*-Min forest. We are now ready to present the main procedure of phase 1, Find-k-Min, which is given in Figure 3.4. (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_0 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$(F + F')$

FIG. 3.4. *The Find-k-Min procedure.*

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). Lemmas 3.9 and 3.10 establish the correctness of this procedure. The performance of Find-k-Min is analyzed in section 3.5.

LEMMA 3.9. *Let H' be a graph formed by sampling each edge in H with probability p, and let F be a k_0-Min forest of H' (derived by at least $\log(k_0 + 1)$ Borůvka steps). The call to Filter(H, F) returns a graph containing a k_0-Min forest of H, whose expected number of edges is no more than n/p.*

Proof. By Claim 3.1, any edge in the k_0-Min forest of H is k_0-Min-light w.r.t. H'. By Lemma 3.8, no edges k_0-Min-light w.r.t. H' are filtered; this establishes the first part of the lemma. By the second part of Lemma 3.8, all edges not filtered are k-Min-light w.r.t. H' for some k. According to Lemma 3.1, the number of edges in H that are k-Min-light w.r.t. H' for any k is no more than n/p. This establishes the rest of the lemma. □

LEMMA 3.10. *The call Find-k-Min$(G_t, \log^* n)$ returns a set of edges that includes the k_0-Min tree of each vertex in G_t.*

Proof. The proof is by induction on i. For $i = 3$ (the base case) Find-k-Min$(H, 3)$ returns F, which by Lemma 3.3 contains the k_0-Min tree of each vertex. Now assume inductively that Find-k-Min$(H, i - 1)$ returns the k_0-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_0-Min tree of each vertex in H_s. By Lemma 3.9, the call to Filter(H_c, F_s) returns in H_f a set of edges that contains the k_0-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_0-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_0-Min tree of each vertex in H. □

3.5. Performance of find-k-Min. In this section we bound the time and work required by the Find-k-Min procedure.

CLAIM 3.7. *The following invariants are maintained at each call to Find-k-Min. The number of live vertices in* $H \le n/(\log^{(i)} n)^4$*, and the expected number of edges in* $H \le m/(\log^{(i)} n)^2$*, where* m *and* n *are the number of edges and vertices in the original graph.*

Proof. These hold for the initial call, when $i = \log^* n$. By Lemma 3.3, the contracted graph H_c has $\le n/(\log^{(i-1)} n)^4$ 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 $\le m/(\log^{(i-1)} n)^2$, maintaining the invariants for the first recursive call.

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

LEMMA 3.11. *Find-k-Min*$(G_t, \log^* n)$ *runs in expected* $O(\log n)$ *time and* $O(m)$ *work.*

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 graph, 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 ${k_0}^{k_0^{O(\log^{(3)} n)}}$, thus the time needed for each modified Borůvka step and each Filter step is ${k_0}^{O(\log^{(3)} n)}$. 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. By Claim 3.7, the expected number of edges in an invocation at level i is $O(m/(\log^{(i)} n)^2)$. Since there are $O(\log^{(i)} n)$ Borůvka steps performed in this invocation, the work required is $O(m/\log^{(i)} n)$. There are $2^{\log^* n - i}$ invocations with depth parameter i; therefore 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 $\le n/k_0$)
$G_s :=$ Sample edges of G' with prob. $1/\sqrt{k_0} = 1/\log^{(2)} n$
$F_s :=$ 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 $O(\frac{m_1}{m} \log n \log^{(2)} n)$ time, where m_1 is the number of edges in G_1.

The graphs G_s and G_f each have expected $m/\sqrt{k_0} = m/\log^{(2)} n$ edges since G_s is derived by sampling each edge with probability $1/\sqrt{k_0}$, and, by the sampling lemma of [KKT95], the expected number of edges in G_f is $(m/k_0)/(1/\sqrt{k_0}) = m/\sqrt{k_0}$. Because we call Find-MSF on graphs having expected size $O(m/\log^{(2)} n)$, each call takes $O(\log n)$ time.

4.1. The Find-MSF procedure. The procedure Find-MSF(H), given in Figure 4.1, is similar to previous randomized parallel algorithms except it uses no recursion. Instead, a separate *BaseCase* 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].

As its BaseCase, we use the simplest version of the algorithm of Chong, Han, and Lam [CHL01], which takes time $O(\log n)$ using $(m+n)\log n$ processors. By guaranteeing that it is called only on graphs of expected size $O(m/\log^2 n)$, the running time remains $O(\log n)$ with $(m+n)/\log n$ processors. An adaptation of our algorithm to the CRCW PRAM leads to one roughly twice as fast as [CKT96]. Because of a more efficient phase 1 we can afford to make only four BaseCase calls in phase 2, rather than eight calls as in [CKT96].

Find-MSF(H)
  $H_c :=$ Borůvka-B$(H, \log^4 n, F)$
  $H_s :=$ Sample edges of H_c with prob. $p = 1/\log^2 n$
  $F_s :=$ BaseCase(H_s)
  $H_f :=$ Filter(H_c, F_s)
  $F' :=$ BaseCase(H_f)
  Return$(F + F')$

FIG. 4.1. *The Find-MSF procedure.*

After the call to Borůvka-B, the graph H_c has $< m/\log^4 n$ vertices. Since H_s is derived by sampling the edges of H_c with probability $1/\log^2 n$, the expected number of edges to the first BaseCase call is $O(m/\log^2 n)$. By the sampling lemma of [KKT95], the expected number of edges to the second BaseCase call is $< (m/\log^4 n)/(1/\log^2 n)$, 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, shown in Figure 4.2, returns a contracted version of H with $O(m/l)$ 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 $> l$.

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 as well as the idea of a growth control schedule were introduced by Johnson and Metaxas [JM92].

```
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
```

FIG. 4.2. *The Borůvka-B procedure.*

5. 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, and 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.

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 [AS00], 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 is dominated by 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 probability that phase 2 uses $O(m)$ work is $1-\exp(-\Omega(m/\log^2 n))$. We omit the analysis as it is similar to the analysis for phase 1.

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 scheme given in [HZ96] that takes logarithmic time and linear work overall but uses *superlinear* space. An algorithm claimed in [HZ01] uses linear space; however, it is not given a clear description in [HZ01] and, more seriously, it makes heavy use of a nontrivial linked-list based sorting algorithm of Goodrich and Kosaraju [GK96]. In this section we give a self-contained description of a processor allocation scheme for "tree structured" computations which does not use any sorting subroutine.

Let M be a set of m processes which perform some computation. So long as the computation is *tree structured*, in the sense given below, its exact nature is unimportant. At any point in the computation there is a set $D \subseteq M$ of *dead processes* and a stack $\mathcal{S} = (S_0, S_1, \ldots, S_d)$, where $S_0 = M$ and $S_{j+1} \subseteq S_j$. In the ith round of computation, the stack is potentially changed and some set $R_i \subseteq M$ of the processes compute for t_i time steps. Round i follows these steps:

1. Either (a) $\mathcal{S}$ is unchanged, $R_i := S_d - D$, or
   (b) $\mathcal{S} := (S_0, \ldots, S_d, S_{d+1})$, $S_{d+1} \subseteq S_d$, $R_i := S_{d+1} - D$, or
   (c) $\mathcal{S} := (S_0, \ldots, S_{d-1})$, $R_i := S_{d-1} - D$.
2. The R_i do something for $t_i \geq 1$ steps.
3. $D := D + \{\text{some subset of } R_i\}$.

This is a rather technical characterization of a class of algorithms. Informally, any recursive algorithm fits into this scheme if the active processes in one recursive call are a subset of the active processes from its parent call.

Let $p \leq m$ be the number of EREW processors available. Ideally, we would like to simulate round i in $O(\lceil R_i/p \rceil)$ time (i.e., with zero overhead). Like [HZ96] our overhead is nonconstant but usually negligible.

THEOREM 7.1. *For some tree computation, let r be the total number of rounds, $T = \sum_i t_i$ be the total time for all rounds, $W = \sum_i t_i \cdot |R_i|$ be the total work for all rounds, d_{max} be the maximum depth of the stack, and $q = \Omega(\log(mr))$ be a parameter. Then with probability $1 - e^{-\Omega(q)}$ the computation of m processes can be simulated with p EREW processors in $O(T + W/p + r\log q + \log p)$ time. The space required is $O(m + p \cdot d_{max})$. It is assumed there is some (easily computable) bound r_i such that $r_i \geq |R_i|$ and $r_i = O(|R_i|)$.*

In our MSF algorithm the number of rounds $r = O(2^{\log^* n} k_0 \log k_0) = O(\log^3 \log n)$, the time $T = O(\log n)$, work $W = O(m)$, and $d_{max} = O(\log^3 \log n)$. Plugging these values into Theorem 7.1, our MSF algorithm can be simulated in $O(m/p + \log n + \log q \log^3 \log n)$ time with probability at least $1 - e^{-\Omega(q)}$, using space $O(m + p \log^3 \log n)$. Since $p < m/\log n$, the space is linear in m. We could set $q =$

$\Theta(\log(mr)) = \Theta(\log n)$ and achieve a polynomially small error probability or set q as high as $2^{\log n/\log^3 \log n}$ for a much smaller error probability. Also, the "dead processes" correspond to those edges known to be k_0-Min-heavy (in phase 1) or not in the MSF at all (in phase 2).

As in [HZ96] we organize the processes into *blocks* of size $b = qm/p$ (q processors per block) as follows. We imagine placing the processes deterministically into an $m/b \times b$ array, then performing a random rotation on each column. The processes that end up in the same row are in the same block. Computing this initial allocation is easily done in $O(m/p + \log p)$ time. Since processors from different blocks do not communicate we will isolate our discussion to a single arbitrary block. Let B denote the set of processes in this block; initially $|B| = b$.

We maintain the invariant that the block is represented as a linked list $L = L_d, L_{d-1}, \ldots, L_0$, where $L_d = S_d - D$, and, in general, $L_j = S_j - S_{j+1} - D$. That is, $L = B - D$: no dead processes appear in this list, and L_j lists those processes that do not appear higher up in the stack. We also maintain that for all j, L_j has been fairly allocated. What this means is that the ℓth processor ($0 \le \ell < q$) assigned to this block "owns" a sublist $L_{\ell,j}$ of L_j extending from element $\ell\lceil \frac{|L_j|}{q} \rceil$ to element $(\ell+1)\lceil \frac{|L_j|}{q} \rceil - 1$ (if they exist). We assume processor ℓ has a pointer to $L_{\ell,j}$. (These pointers contribute the pd_{max} term to the space in Theorem 7.1. The other space requirements are linear in m.)

Suppose in round i, step 1 is of type (a)—the stack is not altered. Then $R_i \cap L = L_d$ and we already have a fair allocation of L_d. Provided $|L_d|$ is about the same in this block as in any other, step 2 can be simulated optimally in $O(t_i \cdot \lceil |L_d|/q \rceil)$ time. This will be discussed later. To restore our invariants after step 3 we simply need to splice out newly dead processes from L_d and compute a fair allocation for the new list. Let L_d and L'_d be the list before and after step 3. Processor ℓ will find all $L'_{w,d}$ which lie in $L_{\ell,d}$, sending a pointer of $L'_{w,d}$ to processor w. For this task processor ℓ must know $|L'_d|$ and the number of elements from L'_d which lie before $L_{\ell,d}$, both of which can be computed in $O(|L_d|/q + \log q)$ time with a prefix-sums computation. Finally we compute L'_d by splicing out all dead elements, also in $O(|L_d|/q + \log q)$ time.[2] The other two cases for step 1, (b) and (c), involve either splitting L_d into two lists or combining L_d and L_{d-1} into one list, followed by a step to compute a fair allocation for the new list(s). We omit a discussion of these two cases; the techniques used are the same as in step 3.

In implementing step 2 we use the assumption that there is a known upper bound $r_i \ge |R_i|$ on the number of processes taking part in the ith round. (In our MSF algorithm, for instance, this upper bound would hold w.h.p.) We argue that with a certain probability (that depends on q) for every round i, every processor is given no more than $(1+\epsilon)(1 + r_i/p)$ active processes. Each of the t_i time steps in step 2 is then easily simulated in $(1+\epsilon)(1 + r_i/p)$ time. Consider the $m/b \times b$ array used in the initial allocation, and an arbitrary block and round. Let X_k be 1 if the process initially placed in the kth column is active in the round, and 0 otherwise. Because the rotations on different columns were *independent*, so too are the X_k's. Let $X = \sum_{k=1}^{q} X_k$ be the number of active processes appearing in the block; clearly $E(X) = |R_i|b/m \le r_i b/m$. Since each processor can be thought to have a "dummy" process associated with it which is active in every round, assume, without loss of generality, that $E(X) \ge q$. Noting that X is the sum of independent Bernoulli trials,

[2] The prefix-sums and splicing can, of course, be performed in one pass.

we can bound the probability that X deviates too far from its expectation using a Chernoff bound [AS00]. For $0 < \epsilon < 1$, $\Pr[X > (1+\epsilon)E(X)] < e^{-\Omega(\epsilon^2 E(X))}$, and for constant ϵ, the probability that *any* block in any round gets more than $1+\epsilon$ times its expectation is $< \frac{mr}{b} e^{-\Omega(q)} = e^{-\Omega(q)}$ since $q = \Omega(\log(mr))$. The analysis of our scheme is very similar to that of [HZ96] but considerably more efficient in terms of time. In [HZ96] ϵ is increased in order to reduce the probability of failure. In our scheme we would set ϵ to be a small constant and increase q (number of processors per block) as necessary. It is crucial to keep ϵ small because in either scheme nearly all processors spend an $\epsilon/(1+\epsilon)$ fraction of their time doing nothing! On the other hand, the q parameter can usually be increased dramatically with negligible effects on the overall running time. Hence our scheme achieves a low failure probability without excessive processor idling.

We remark that the space claimed in Theorem 7.1 can be reduced to $O(m)$ at the expense of a slightly more complicated scheme. The idea is to compute fair allocations only when necessary. Very frequently, a previously computed fair allocation is "fair enough." For instance, in step 1(b) L_d is split into two lists, L'_d and L'_{d+1}. If L'_{d+1} contains most of the elements from L_d, we might as well use the fair allocation of L_d instead of computing new ones for L'_{d+1} and L'_d.

8. Adaptations to other practical parallel models. Our results imply good MSF algorithms for the QSM [GMR99] 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 EREWcomputations on to QSM given in [GMR99]. Theorem 8.2 follows from the QSM to BSP emulation given in [GMR99] 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.*

9. Conclusion. We have presented a randomized algorithm for MSF on the EREW PRAM which is provably optimal both in time and work. Our algorithm works within the stated bounds with high probability in the input size and has good performance in other popular parallel models.

One drawback to our algorithm is that it uses a linear number of random bits. A recent MSF algorithm [PR02a] for the EREW PRAM performs linear work but uses only a polylogarithmic number of random bits; however, the time required is suboptimal ($O(\log^2 n \log^* n)$). Unlike the algorithm presented here, the [PR02a] algorithm is not a parallelization of [KKT95] and does not use the sampling lemma from [KKT95].

An open question is how to obtain a *deterministic* time-work optimal MSF algorithm. Pettie and Ramachandran [PR02b] have given a provably optimal *sequential* MSF algorithm; however, its exact complexity (and therefore the complexity of MSF) is still unknown. Parallelizing this optimal sequential algorithm seems very difficult.

REFERENCES

[AS00] N. Alon and J. Spencer, *The Probabilistic Method*, 2nd ed., Wiley-Interscience, New York, 2000.

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

[CHL01] K. W. Chong, Y. Han, and T. W. Lam, *Concurrent threads and optimal parallel minimum spanning trees algorithm*, J. ACM, 48 (2001), pp. 297–323.

[CKT94] R. Cole, P. N. Klein, and R. E. Tarjan, *A linear-work parallel algorithm for finding minimum spanning trees*, in Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures (SPAA'94), Cape May, NJ, ACM, 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 Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures (SPAA'96), Padua, Italy, ACM, pp. 243–250.

[Dij59] E. W. Dijkstra, *A note on two problems in connexion with graphs*, 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., 20 (1991), pp. 1046–1067.

[GMR99] P. B. Gibbons, Y. Matias, and V. Ramachandran, *Can a shared-memory model serve as a bridging model for parallel computation?*, Theory Comput. Syst., 32 (1999), pp. 327–359.

[GK96] M. T. Goodrich and S. R. Kosaraju, *Sorting on a parallel pointer machine with applications to set expression evaluation*, J. ACM, 43 (1996), pp. 331–361.

[HZ96] S. Halperin and U. Zwick, *An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM*, J. Comput. System Sci., 53 (1996), pp. 395–416.

[HZ01] S. Halperin and U. Zwick, *Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems*, J. Algorithms, 39 (2001), pp. 1–46.

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

[JM97] D. B. Johnson and P. Metaxas, *Connected components in $O(\log^{3/2} n)$ parallel time for CREW PRAM*, J. Comput. System Sci., 54 (1997), pp. 227–242.

[K97] V. King, *A simpler minimum spanning tree verification algorithm*, Algorithmica, 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*, J. ACM, 42 (1995), pp. 321–328.

[KPRS97] V. King, C. K. Poon, V. Ramachandran, and S. Sinha, *An optimal EREW PRAM algorithm for minimum spanning tree verification*, Inform. Process. Lett., 62 (1997), pp. 153–159.

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

[PR97] C. K. Poon and V. Ramachandran, *A randomized linear work EREW PRAM algorithm to find a minimum spanning forest*, in Algorithms and Computation (Singapore, 1997), Lecture Notes in Comput. Sci. 1350, Springer-Verlag, Berlin, 1997, pp. 212–222.

[PR98] C. K. Poon and V. Ramachandran, *private communication*, 1998.

[PR02a] S. Pettie and V. Ramachandran, *Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms*, in Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, PA, 2002, pp. 713–722.

[PR02b] S. Pettie and V. Ramachandran, *An optimal minimum spanning tree algorithm*, J. ACM, 49 (2002), pp. 16–34.

[Prim57] R. C. Prim, *Shortest connection networks and some generalizations*, Bell System Technical J., 36 (1957), pp. 1389–1401.

[Val90] L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, 33 (1990), pp. 103–111.