Improving the Efficiency of Parallel Minimum Spanning Tree Algorithms[*]

Ka Wong Chong[†] *Yijie Han*[‡] *Yoshihide Igarashi*[§] *Tak Wah Lam*[¶]

Abstract

This paper presents results which improve the efficiency of parallel algorithms for computing the minimum spanning trees. For an input graph with n vertices and m edges our EREW PRAM algorithm runs in $O(\log n)$ time with $O((m+n)\sqrt{\log n})$ operations. Our CRCW PRAM algorithm runs in $O(\log n)$ time with $O((m+n)\log\log n)$ operations. We also show that for dense graphs we can achieve $O(\log n)$ time with $O(n^2)$ operations on the EREW PRAM.

Key words: Parallel algorithms, graph algorithms, minimum spanning trees, PRAM.

1. Introduction

Let $G = (V, E)$ be the input graph, where V is the vertex set and E is the edge set. Also let $|V| = n$ and $|E| = m$ and each edge is assigned a real-valued weight. The minimum spanning tree problem is to compute a spanning tree for each connected component of the graph such that the total weight of the edges in the spanning tree is minimum. For our purpose (and without loss of generality) we assume that each edge is assigned a distinct weight so that the minimum spanning trees are unique. We also assume that the input graph is connected, otherwise our algorithm will find a minimum spanning forest.

Computing minimum spanning trees is a fundamental problem and it has been studied by many researchers [3, 4, 5, 6, 10, 15, 16, 17, 18, 21]. In this paper we consider the problems of computing the minimum spanning trees on the CRCW and EREW PRAM models.

A PRAM is a parallel machine with p processors and a global shared memory. Each processor can perform the usual computation of a sequential machine. Depending on the way the processors access the global memory, PRAM can be classified as EREW (Exclusive Read Exclusive Write) PRAM, CREW (Concurrent Read Exclusive Write) PRAM and CRCW (Concurrent Read Concurrent Write) PRAM. In this paper we use the ARBITRARY CRCW PRAM in which an arbitrary processor succeeds in writing in the case several processors write into a memory cell simultaneously. It is known that CREW PRAM is more powerful than EREW PRAM [19] and CRCW PRAM is more powerful than CREW PRAM [9].

[*]A preliminary version of this paper was presented at the 5th International Conference on Computing and Combinatorics (COCOON'99), Lecture Notes in Computer Sciecne 1627, 452-461(1999).

[†]Department of Computer Science, The University of Hong Kong, Hong Kong, China. Email: kwchong@cs.hku.hk

[‡]Computer Science Telecommunications Program, University of Missouri — Kansas City, 5100 Rockhill Road Kansas City, MO 64110, USA. Email: han@cstp.umkc.edu. http://welcome.to/yijiehan

[§]Department of Computer Science, Gunma University, Kiryu, 376-8515 Japan. Email: igarashi@comp.cs.gunma-u.ac.jp

[¶]Department of Computer Science, The University of Hong Kong, Hong Kong, China. Email: twlam@cs.hku.hk

The best previous deterministic CRCW minimum spanning tree algorithm runs in $O(\log n)$ time with $O((m+n)\log n)$ operations (time processor product) [3, 21] (These two algorithms use different version of the CRCW PRAM model. The algorithm in [21] uses a weaker model than the one used in [3]). For a period the $O(\log^2 n)$-time algorithm was the best known one on the CREW and EREW PRAM for the spanning tree problem [4, 11, 15]. Johnson and Metaxas were the first to show an EREW PRAM algorithm with time complexity $O(\log^{3/2} n)$ and $m+n$ processors (therefore $O((m+n)\log^{3/2} n)$ operations) for computing minimum spanning trees [16]. Johnson and Metaxas' result for minimum spanning trees [16] was imporved by Anderson, Beame and Sinha (private communication) and Chong [5] to time complexity $O(\log n \log^{(2)} n)$ with $O((m+n)\log n \log^{(2)} n)$ operations ($\log^{(1)} n = \log n$, $\log^{(c)} n = \log\log^{(c-1)} n$). Recently Chong, Han and Lam [6] presented an EREW minimum spanning tree algorithm with $O(\log n)$ time complexity and $O((m+n)\log n)$ operation complexity. There are also results of randomized algorithms for the minimum spanning tree algorithm. Karger [17] obtained a randomized algorithm using $O(\log n)$ time and super-linear operations on the EREW PRAM model. Poon and Ramachandran [18] gave a randomized algorithm on the EREW PRAM model using linear expected operations and $O(2^{\log^* n}\log n \log\log n)$ expected time.

In this paper we improve the efficiency of previous results for computing minimum spanning trees. We present an EREW algorithm for computing minimum spanning trees with time complexity $O(\log n)$ using $O((m+n)\sqrt{\log n})$ operations. For dense graphs our algorithm runs in $O(\log n)$ time with $O(n^2)$ operations on the EREW PRAM. We also show that our algorithm runs on the CRCW PRAM with time complexity $O(\log n)$ and operation complexity $O((m+n)\log\log n)$. We achieve our results by applying fast parallel integer sorting [2, 13, 14], parallel graph reduction [4], parallel selection [8] and parallel approximate compaction [12].

2. Background

2.1. Preliminaries

Given an undirected graph $G=(V,E)$, we assume that vertices in V are labeled with integers $\{1,2,...,n\}$. Every vertex v is associated with an adjacency list $L(v)$. Each edge (u,v) is represented twice, once in the adjacency list of u (denoted $<u,v>$) and once in the adjacency list of v (denoted $<v,u>$). We call $<v,u>$ the dual of $<u,v>$. These two representations are linked to each other. The weight of (u,v) is denoted $w(u,v)$. Given a connected subgraph $G_i=(V_i,E_i)\subset G=(V,E)$, an internal edge is an edge $(u,v)\in E_i$ with both $u,v\in V_i$. An external edge is an edge $(u,v)\in E-E_i$ with one of its endpoints belongs to V_i and the other belongs to $V-V_i$. Let $G_j=(V_j,E_j)$ be another connected subgraph of G such that $G_i\cap G_j=\phi$. Distinct edges $(u,v),(x,y)\in E-(E_i\cup E_j)$ having one endpoint in V_i and the other endpoint in V_j are called multiple edges. As computation proceeds, components will be formed. Each component contains several vertices. Thus internal and multiple edges will be generated. We need to remove internal edges and multiple edges (but keep the one with the minimum weight among the multiple edges).

In this paper we give a fine control over the internal and multiple edges. We use new ideas such as the application of parallel integer sorting [2, 13, 14], parallel graph reduction [4], parallel selection algorithm [8] and parallel approximate compaction [12] to remove internal and multiple edges and to find minimum edges. These ideas enable us to improve on the operation complexity (or the processor complexity) of previous best algorithms.

A tree-loop is a directed connected graph in which each vertex has outdegree 1. A tree-loop has exactly one loop (or cycle) in it. Known parallel algorithms compute minimum spanning trees by using the *hook-and-contract* approach. In this approach each vertex first chooses an edge of minimum weight from its adjacency list to hook to its neighbor. These edges used for hooking form a forest of tree-loops. In our case because each vertex chooses the minimum weight edge for hooking, the loop in each tree-loop has exactly two edges in it. And these two edges must be the dual edge of each other. Such tree-loops can be easily converted to rooted trees by breaking the loops in the tree-loops. Then each tree contracts to a representative vertex. The representative vertex is called the supervetex. This process is repeated until minimum spanning trees are computed. Our algorithm also uses this approach.

In previously known parallel algorithms, the hooking operation is done by first computing the minimum external edge for each supervertex and use this minimum external edge to hook to a different supervertex. Because each supervertex may have $O(n)$ edges, the computation of the minimum external edge takes $O(\log n)$ parallel time resulting in $O(\log^2 n)$ time for computing the minimum spanning tree. We shall call this *the long edge list problem.* In order to achieve $O(\log n)$ time we have to overcome this problem and achieve constant time for computing the minimum external edge. Even when we have a short edge list we may still face the second problem namely *the internal and multiple edge problem.* That is, many edges in the edge list are internal and multiple edges. Internal edges prevents us from finding an external edge quickly because if it turns out that the minimum edge we computed is an internal edge then the edge cannot be used for hooking. Multiple edges do not prevent hooking at the current stage but they will become internal edges later on and create problems for later hooking operations. We will show how to overcome these two problems.

2.2. Basics

Our algorithm has $\log n$ stages. For the moment we assume that each stage takes constant time. Let B_i be the set of minimum spanning tree edges computed at the end of stage i. B_i induces a set of trees $F_i = \{T_1, T_2, ..., T_k\}$. We maintain the property that each T_j, $1 \leq j \leq k$, contains at least 2^i vertices. At stage $i+1$, the minimum external edge for each tree in $F_i' \subseteq F_i$ is computed and used to hook trees in F_i together to form larger trees. F_i' is a subset of F_i guaranteed to contain all trees in F_i of size $< 2^{i+1}$. Thus at the end of stage $i+1$ each tree in F_{i+1} contains at least 2^{i+1} vertices. A tree in F_i of size $\geq 2^s$, $s > i$, is said grown for stage s.

2.2.1. Basics about Active Lists

To overcome the long edge list problem we form a partial list containing the $2^s - 1$

minimum edges from the adjacency list, where s denotes the number of stages we intend to grow the minimum spanning tree. We form a graph where each vertex has only a partial adjacency list containing $2^s - 1$ minimum edges from its original adjacency list (in the case where the adjacency list of a vertex has less than $2^s - 1$ edges then all the edges in the adjacency list will be in the partial adjacency list). We call this partial list the active list of the vertex, and edges adjacent to the vertex not in the active list are in the inactive list. Edges in the active list are active edges while edges in the inactive list are inactive edges. Note that it is possible that $< u, v >$ is an active edge of u while $< v, u >$ is an inactive edge of v. In this case (u, v) is a half edge. We call $< u, v >$ a half edge and $< v, u >$ a dual half edge. When both $< u, v >$ and $< v, u >$ are active edges (u, v) is called a full edge. When both $< u, v >$ and $< v, u >$ are inactive edges (u, v) is called a lost edge. Full edges are needed when we will combine the active lists of several supervertices into the active list of one supervertex. Half edges cannot be used to combine the active lists of supervertices, but they merely indicate that hooking takes place among supervertices. Lost edges are used neither for hooking nor for combining supervertices.

For each vertex v we define a threshold $h(v)$. If all edges incident to v are in active list then $h(v) = \infty$. Otherwise $h(v) = w(e_0)$, where e_0 is the external edge of minimum weight in v's inactive list. For a set S of vertices we let $h(S) = \min\{h(u)|u \in S\}$. An active list is clean if it does not contain internal edges (full or half) and full multiple edges. It is possible that a clean active list contains a full external edge e and several half edges which are multiples of e. Fig. 1 demonstrates this situation.

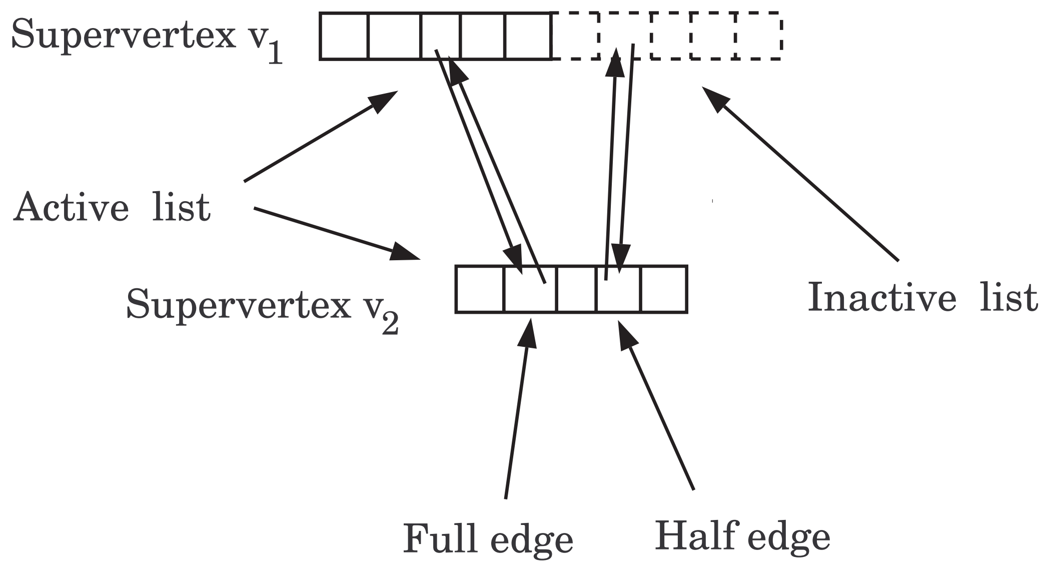


Fig. 1. Full edges and half edges.

When active lists are used some edges are left out. Will it happen that the minimum external edge needed for hooking will be left out also? Note that this will not happen initially, but it may happen in the later stages. We now analyze under what situations this is guaranteed not to happen.

Below we will analyze the active lists at stage $i+j$, where stage i is the stage when our computation starts with clean active lists. Stage $i+j$ represents any stage after stage i. We shall also present the active list construction at stage $i+j$ and then analyze the constructed active lists at stage $i+j+k$, i.e. any stage after stage $i+j$.

Suppose that a clean active list for each vertex (we use vertex instead of supervertex here because we will use supervertex for later stages) is constructed at the end of stage i. Denote this set of active lists by A. Suppose that each active list in A contains 2^s-1 edges (or all edges if the number of incident edges is no more than 2^s-1). The essence of the following lemma is given in [16].

Lemma 1: At any stage $i+j$, $1 \leq j < s$, for each supervertex v either the minimum external edge of v is in a list in A or v contains at least 2^s vertices.

Proof: We analyze two cases. Case 1 is the situation where no edges are left out of the active lists and case 2 is the situation that some edges are left out.

Case 1. The thresholds for all vertices in v are ∞. In this case all edges incident to v are active edges. Thus either there is a minimum external edge in a list in A for v or v is isolated. Because we assume G is connected therefore v is isolated means that the minimum spanning tree for G is computed.

Case 2. There is a vertex in v whose threshold is not ∞. Let u be the vertex in v which has the smallest threshold $h(u)$. Consider the set S of edges incident to v which has weight $< h(u)$. All these edges are active edges. The set S' of all active edges incident to u is in S. If not all edges in S are internal edges of v, then we simply pick the minimum external edge in S and it is the minimum external edge of v. If all edges in S are internal edges of v, then all edges in S' are internal edges as well. If all edges in S' are full edges then the edges in S' are incident to 2^s vertices. And therefore v contains at least 2^s vertices. S' cannot contain a half internal edge. Let $< u, z >$ be such an edge. Then we have $h(z) \leq w(u,z) < h(u)$ which is a contradiction because we assumed that $h(u)$ is the smallest. $\square$

The clean active list containing at least 2^s-1 edges (or all incident edges) is called a (2^s-1)-active-list. Lemma 1 basically says that if vertex v's (2^s-1)-active-list is exhausted (all edges in it becoming internal edges) then v and v's 2^s-1 neighboring vertices are all in the supervertex containing v. Thus clean (2^s-1)-active lists will enable us to grow supervertices so that each supervertex contains at least 2^s vertices.

Because of Lemma 1 we will maintain the following property for our algorithm:

Property 1: In stage $i+j$, $1 \leq j < s$, only edges in A are used for hooking.

In fact we can strengthen Property 1. The observation [16] is as follows. When a tree T is computed, each vertex v in T provides a threshold $h(v)$. Let v' provides the smallest threshold $h(v')$. Then by the essence of the proof of Lemma 1, if all edges with weight $w < h(v')$ have become internal edges then T contains at least 2^s vertices. Thus if T contains less than 2^s vertices then we can always find the minimum external edge among edges of weight $< h(v')$. For any $j < s$ let T be a tree at stage $i+j$. Let $u \in T$ be the vertex with the smallest threshold among vertices in T. Let S be the set of active edges incident to T with weight $< h(u)$. Then either the minimum external edge is in S or T contains at least 2^s vertices. Thus we maintain

Property $1'$: In stage $i+j$, $1 \leq j < s$, only edges in S are used for hooking.

When a tree T uses a half edge e to hook to another tree T_1, e is the minimum external edge of T while $w(e)$ is no less than $h(u) = \min\{h(w)|w \in T_1\}$. Because of Property $1'$ we need only edges of weight $< h(u)$ to grow trees. Now $w(e) \geq h(u)$ and e is the minimum external edge of T, therefore all the edges we need to grow $T \cup T_1$ are in the active lists of vertices in T_1. Thus we maintain the following property for our algorithm.

Property 2: If a tree T uses a half edge to hook to another tree, then T and all vertices and edges incident to T will be labeled as inactive. Inactive signals that vertices and edges do not need to participate in the minimum spanning tree computation before stage $i+s+1$.

For a particular j for each tree T computed at stage $i+j$ we build a new clean active list L_T which contains the minimum $2^{s-j}-1$ edges e incident to T with $w(e) < \min\{h(u)|u \in T\}$. It is obvious that edges in L_T are in A. We also update the threshold for each new active list constructed. Let us use A_1 to denote the set of all (edges in) L_T's constructed for supervertices. By Lemma 1 at any stage $i+j+k$ with $j+k < s$ for each supervertex v either the minimum external edge is in A_1 or v contains at least 2^s vertices. Property 1 is refined to read: In stage $i+j+k$, $j+k < s$, only edges in A_1 are used for hooking.

It is possible that half edges are used for hooking. Because of Property 1 it is not possible that dual half edges or lost edges are used for hooking.

When a set S of vertices are hooked together with full edges to form a supervertex we can form a linked cycle containing all edges in the active lists of vertices in S, as shown in Fig. 2. This linked cycle then enables us to group the active lists of vertices in S into one list. When half edges are used for hooking we cannot form a linked cycle, see Fig. 2. The half edge is where it breaks.

2.2.2. More Involved Analysis for Active Lists

We need the notion of root component. When every vertex uses the minimum external edge for hooking, we obtain a set of tree-loops. The loop in each tree-loop contains two edges which are dual of each other. By breaking the loop and choosing an arbitrary vertex in the loop to become the root we obtain a rooted tree. Initially each vertex is its own root component. When minimum external edges are used to hook vertices together to form a rooted tree the fully connected component is a maximal set of vertices hooked together by full edges. The root component of the tree is the fully connected component containing the root of the tree. A nonroot component is a fully connected component not containing the root of the tree. We also say that the active edges incident to vertices in a fully connected component are in that component. Fig. 3 illustrates root component and fully connected components.

Lemma 2: Let T' be the root component of T. Then $h(T') = h(T)$. Also the minimum external edges of T with weight $< h(T)$ are all in the root component.

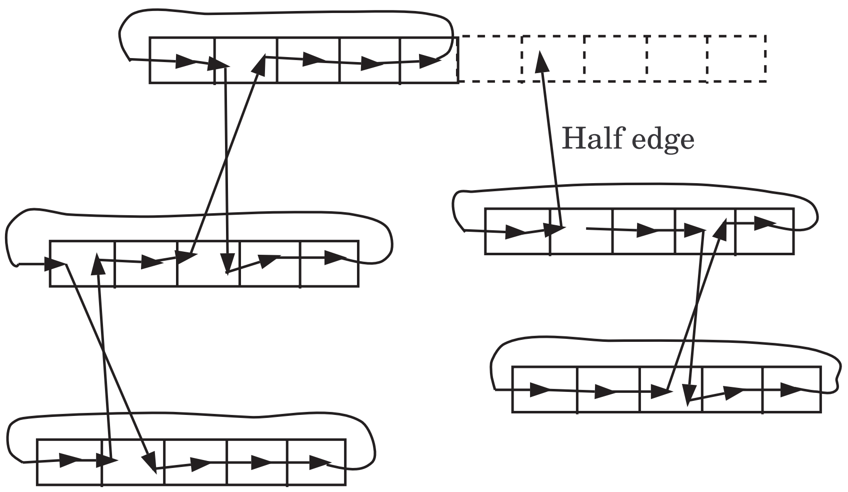


Fig. 2. Merge active lists.

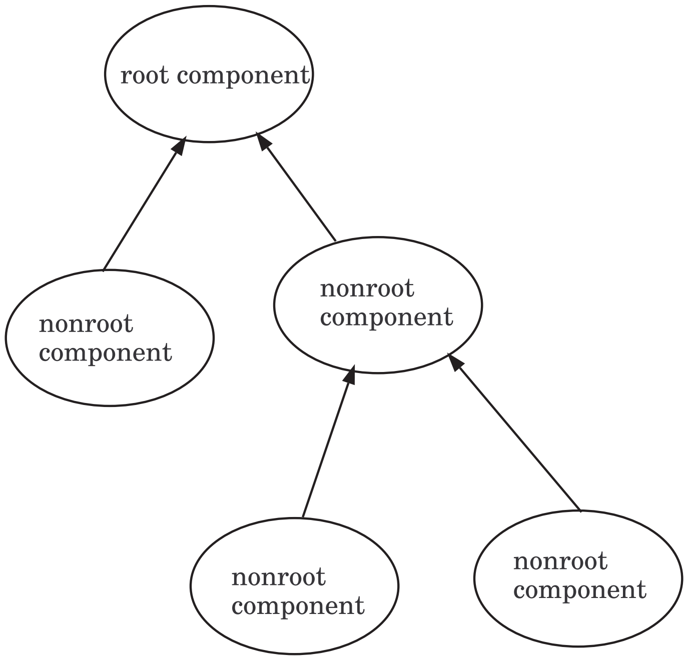


Fig. 3. Root component and nonroot components.

Proof: Consider each tree T. If we contract each fully connected component in T into a vertex then these vertices and the half edges used for hooking form a directed rooted tree R. Therefore we obtain the parent-child relation for fully connected components in T. For each fully connected component C in T we use $e(C)$ to denote the external half edge which hooks C to C's parent. For each nonroot component C we can use induction to show that the weight of $e(C)$ is less than the threshold of any vertex in C. By Property $1'$ this is true at the stage when $e(C)$ is computed as the minimum external edge used for hooking. Because $e(C)$ is a half edge C becomes inactive after hooking. If in a later stage another tree T_1 uses a full edge e_1 hooking into C, by induction hypothesis we have $w(e_1) < h(C_1)$, where C_1 is the fully connected component in T_1 which is to be merged with C by the hooking using edge e_1. Therefore we have $w(e(C)) < w(e_1) < h(C_1)$. Therefore the weight of $e(C)$ maintains to be less than $h(C \cup C_1)$.

Thus the weight of $e(C$'s parent) is less than the weight of $e(C)$. Therefore $e(C)$ must be the minimum external edge of the tree rooted at C in R. Thus we have that for any nonroot component C $h(C) > w(e(C)) > h(T')$ and $h(T') = h(T)$. We also conclude that the minimum external edges of T with weight $< h(T)$ are all in the root component. $\Box$

By Lemma 2 our algorithm uses only minimum external edges in the root component for hooking.

When half edges as well as full edges are used for hooking, the tree T we obtained can be decomposed into a set S of fully connected components. Because of Lemma 2 all vertices and edges in nonroot components should be labeled as inactive.

Now we address the internal and multiple edge problem. Internal and multiple edges will be generated as supervertices are formed. Therefore we have to perform compression repeatedly to remove internal and multiple edges. Compression is done by first sorting the edges so that multiple edges are consecutive in the adjacency list, then removing internal and multiple edges and then packing the remaining edges to consecutive positions in the adjacency list.

Let us consider the situation that edges in lists in A in Lemma 1 are compressed at the beginning of stage $i + j + 1$, $j < s$. If no more than 2^s vertices are hooked together all by full edges to form a supervertex, then the active lists of these vertices can be merged into one list L and the compression of the edges on L takes $O(s)$ time because there are no more than 2^{2s} edges to be sorted. In other words, if we expend $O(s)$ time and cannot finish compression then the tree we computed contains more than 2^s vertices. In this case the tree is grown for stage $i + s$ and we mark all edges incident on this tree inactive. Inactive means that before stage $i + s + 1$ nothing needs to be done with the vertices and the edges.

By applying compression each vertex in each nonroot component will find out that either it is in a fully connected component containing at least 2^s vertices when compression cannot finish in $O(s)$ time, or it is hooked to an inactive vertex, or it is hooked to another fully connected component by a half edge. In either of these cases all vertices and edges in the nonroot component can be labeled as inactive. The root component in S does not hook to another fully connected component in S although other nonroot component in S may hook to the root component.

If we cannot finish compression in $O(s)$ time for active edges in the root component then all edges incident to the root component will be marked as inactive. In this case the tree is grown and contains at least 2^s vertices. If we finish compression in $O(s)$ time then all full internal and full multiple edges of the root component will be removed. In this case we build the active list for the supervertex v representing T as follows. First compress edges in the root component C. We then compute $h(C)$ and then all active edges of the root component whose weight is no less than $h(C)$ will be moved to the inactive list. Among the remaining edges in the root component we then pick the minimum $2^{s-j}-1$ edges (note that all full internal and full multiple edges have already been removed) and form the active list of v. We also update the threshold of the new active list. This process is called the active list construction.

Lemma 3: The constructed active list of v is clean.

Proof: By construction the active list does not contain full internal edges and full multiple edges. Hence to prove that it is clean it suffices to prove that the list does not contain half internal edges.

Let $< u, w >$ be such a half internal edge. For $< u, w >$ to be active u must be in the root component of tree T. Consider the cycle C formed by $< u, w >$ and T. If all edges of C (including (u, w)) are full edges at stage i then edge (u, w) would have been detected as an internal edge and removed during compression. If all edges in C except $< u, w >$ are full edges at stage i then $w(u, w) \geq h(w) \geq h(v)$. Therefore $< u, w >$ would have been moved to inactive list during active list construction for v. Otherwise let $< x, y >$ be the half edge at stage $i + j$ which is closest to u in $C- < u, w >$. We have $w(u, w) > w(x, y)$ (because (u, w) is a nontree edge while (x, y) is a tree edge) $\geq h(y)$ (because $< x, y >$ is a half edge) $\geq h(v)$ (because y and u are in the root component). This contradicts the assumption that $< u, w >$ is in the active list after active list construction. □

We denote the set of active lists constructed by A_1.

By Lemma 1 at stage $i + j + k$ either the minimum external edge of a supervertex v is in A_1 or v contains at least 2^s vertices.

Referring to Lemmas 1 and 3 we say that A (A_1) can be used to grow supervertices to contain 2^s vertices.

If in the active list construction we keep only the minimum $2^{s-j-t} - 1$ noninternal and non-full multiple edges in each active list constructed, then these new active lists can be used to grow supervertices to contain 2^{s-t} vertices.

In our algorithm hooking takes constant time and compression can start after hooking is done. Let the hooking be done in stage i and the compression starts at stage i. Suppose $r \leq 2^s$ vertices are hooked into a supervertex v. Let S_1 be the set of active edges incident to supervertex v at stage i. Since S_1 is formed after hooking, it contains no more than $r2^s$ edges because r vertices are hooked into the supervertex. S_1 contains internal and multiple edges at stage i. Let $S_2 \subset S_1$ be the set of edges which remain after active list construction. S_2 does not contain internal and full multiple edges if the compression is done before the next hooking takes place (we say that S_2 is clean at stage i).

Note that if we overlap the compression with several stages, then when the compression is done after $O(s)$ time several (cs for a constant c) stages of hooking already have happened and therefore S_2 contains newly generated internal and multiple edges within these cs stages (we say that S_2 is not clean at stage $i+cs$). Since before the hooking the active list of each vertex has $2^s - 1$ edges, it is guaranteed that these edges can be used to grow the vertex to a supervertex containing up to 2^s vertices. Thus if we finish compression within $s/4$ stages which is used to grow the supervertex to contain $2^{s/4}$ vertices, the edges in S_2 can be used to grow the supervertex further until it contains 2^s vertices. Note that S_2 has no internal and multiple edges at stage i, but it has internal and multiple edges at stage $i+cs$. Therefore the minimum external edge in S_2 cannot be found in constant time at stage $i+cs$. We use the approach described in the next section to solve this problem. The approach given here is a different version of the approach given in [6].

3. An EREW Algorithm with $O(\log n)$ Time and $O((m+n)\log\log n)$ Processors

This section serves the purpose of presenting the basic structure of our algorithm. The optimization given in next section is based on the structure of this section. An alternative version of the algorithm given in [6] is presented here. This version gives a different view and an alternative approach to the minimum spanning tree problem and it affords further improvement as we will present in the next section. The ideas presented in Section 2 provides the facts about active lists and active list construction which are now used in this section.

The main idea behind our algorithm can be outlined as follows. Because we use active lists, at certain stage the minimum external edge may be left out and therefore we have to replenish the active lists. If we replenish the active lists from the inactive lists directly it will take a long time to finish replenishing because there are many edges in the inactive lists, and therefore an $O(\log n)$ time algorithm cannot be achieved. What we do is to form many levels. There are more edges at a lower numbered level than at a higher numbered level. Each level replenishes edges for the next higher numbered level. The observation is that a larger active list takes long time to replenish, but it also takes long time to exhaust (use up) all the edges in the active list. Also note that the hooking operation will create internal and multiple edges in active lists at all levels. Therefore at each level the active lists have to be compressed repeatedly to remove internal and multiple edges. The details of our algorithm are given below.

We will arrange the edge compression into $(1/2)\log\log n$ levels. All levels are executed concurrently. For the moment let us consider a model with one level. Suppose we use one level (level i) for compressing edges. The $\log n$ stages of our algorithm is partitioned into intervals at each level. At level i an interval contains $\log n/4^i$ stages. Within such an interval a tree is grown from size s to size $2^{\log n/4^i}s$. Also within each interval we can compress a list of size at most $2^{c\log n/4^i}$ for a constant c. We take c to be 18. For each vertex we take a $2^{12\log n/4^i}$-active list. Thus if no more than $2^{6\log n/4^i}$ vertices are hooked together then a list of edges of the vertices hooked together will contain no more than $2^{18\log n/4^i}$ edges and we can compress such a list in an interval. If we fail to compress a list in an interval then there are more than $2^{6\log n/4^i}$ vertices hooked together to form the list and we consider the tree grown already. Now consider 4 consecutive intervals. At the beginning of each interval (except for the first interval) we use the minimum spanning tree edges found in the previous interval to hook vertices together. This will give us a list of edges for each fully connected component [5, 16]. Within the time of the interval we compress the edges in each list and construct active list. Note that at the end of an interval I we have the edges compressed and therefore constructed active lists are clean if no new minimum spanning tree edges are computed and no hooking takes place within interval I. In reality within interval I new minimum spanning tree edges are computed and therefore the active lists we obtained are not clean. We say that the active lists constructed at the end of interval I are clean at the beginning of I, but not clean at the end of I. At the end of I the active lists are constructed, but $\log n/4^i$ stages of hooking have already happened in I. Thus we say that the active lists have a lag of $\log n/4^i$ stages. In order to obtain the minimum external edge for each tree in each stage we have to reduce the lag to constant time. This is done by using levels with smaller intervals which will give us small lags. However, within small intervals we cannot handle a large active list. Small active list can only grow trees by a few stages. To replenish edges to small intervals we build $(1/2)\log\log n$ levels. Lower numbered levels have larger intervals and larger lags, and they keep replenishing edges to higher numbered levels which have smaller intervals and smaller lags.

We arrange the edge compression into $(1/2)\log\log n$ levels. Each level is a process. All processes are executed concurrently. When we say process i we mean level i and vice versa. Each level is divided into intervals. An interval at level i contains $\log n/4^i$ consecutive stages. Therefore level 1 has 4 intervals, level 2 has 16 intervals and so on. Each interval at level i overlaps with 4 intervals at level $i+1$. A run of process i has 5 intervals of level i (except for the last run). A run of process i starts at the point where an interval of level $i-1$ begins. Therefore a run of process i always starts at stage $(j-1)\log n/4^{i-1}+1$, $j = 1, 2, 3....$ We call the run of process i which starts at stage $(j-1)\log n/4^{i-1}+1$ the j-th run of process i. We note that the last run has only 4 intervals. Note that because each run has 5 intervals, the last interval of the j-th run at level i overlaps with the first interval of the $(j+1)$-th run at level i. During this overlapped interval two runs of the process execute in parallel. (We may think of each process containing two subprocesses which can run in parallel). The first interval of a run (except for the first run) of process i is called the initialization interval of the run. The remaining intervals of a run are called the effective intervals of the run. The first interval of the first run is also an effective interval. Note that effective intervals for all runs at a level do not overlap. These effective intervals cover the overall $\log n$ stages.

Our intention is to let a run at level i get $(2^{12\log n/4^i})$-active-lists from the active lists at level $i-1$ at the beginning of the run and then repeatedly compress the active lists for 5 times in 5 intervals. There is only one run at level 1. That run gets the input graph at the beginning of the run. At the beginning of each interval the run updates the hooking among active lists with the minimum spanning tree edges found in the stages covered by the previous interval (i.e. the run accomplishes the hooking of supervertices by the minimum spanning tree edges found). Because an interval at level i has $\log n/4^i$ stages the

$(2^{12\log n/4^i})$-active-lists can be compressed in an interval if no more than $2^{6\log n/4^i}$ vertices are hooked together by full edges into the root component of a supervertex. This is to say that if we cannot finish compressing within the interval then the supervertex has already contained enough vertices (the supervertex is already grown for the run) and we need not do compressing for the remaining intervals of the run. In this case the compressing is done in lower numbered levels.

Because a run at level i compresses the edges within an interval, the remaining edges after compressing contains newly generated internal and multiple edges within the interval. If the interval starts at stage t and finishes at stage $t+\log n/4^i-1$, then the active list constructed is clean at stage t but it is not clean at stage $t+\log n/4^i$.We say that the run has a lag of $\log n/4^i$ stages. At the beginning of the first interval of a run at level i, the run constructs a set H_1 of $(2^{12\log n/4^i})$-active-lists from the set H_2 of active lists at the end of an interval I at level $i-1$. Because each of these active lists at level i is obtained from level $i-1$, it has a lag of $\log n/4^{i-1}$ stages. That is, vertices in H_1 have already hooked together to form supervertices such that each supervertex contains $2^{\log n/4^{i-1}}$ vertices. Since H_2 is clean at the beginning of interval I H_1 is also clean at the beginning of interval I. If interval I starts from stage t then active lists in H_1 can be used to grow supervertices to contain at least $2^{t+12\log n/4^{i-1}}$ vertices. That is, edges in H_1 can be used until stage $t+12\log n/4^{i-1}$. Because H_1 is obtained at the end of interval I which contains $\log n/4^{i-1}$ stages, the active lists in H_1 can be used for another $8\log n/4^i$ stages. We construct new $(2^{8\log n/4^i})$-active-lists from the compressed edges by the end of the first interval for the run at level i. This set of active lists is the input to the second interval. At the end of the first interval, the run has a lag of only $\log n/4^i$ stages.

By the same reasoning we construct $(2^{(8-j+1)\log n/4^i})$-active-lists by the end of the j-th interval (which is the input to the $(j+1)$-th interval), $j=2,3,4$. At the end of the 5-th interval the remaining edges are discarded. Note that the first interval of the next run at the same level overlaps with the 5-th interval of the current run. Note that at the beginning of the first(initialization) interval of a run at level i the active lists are obtained from level $i-1$, and it therefore has a lag of $\log n/4^{i-1}$ stages. These active lists are not passed to level i+1 because of their lag. At the beginning of each effective interval the active lists are passed to level $i+1$ (by picking some minimum edges to make smaller active lists for level $i+1$). A run at level i does not pass its active lists to level $i+1$ at the beginning of its initialization interval. At that point the previous run at level i is at the beginning of its 5-th interval and it passes the active lists to level $i+1$.

Because a run at level i has a lag of $\log n/4^i$ stages, a run at level $(1/2)\log\log n$ has a lag of only 1 stage. That is, we can obtain the minimum external edge for a supervertex from level $(1/2)\log\log n$ in constant time. Because each stage grows the supervertex to contain at least two vertices, after $\log n$ stages the minimum spanning tree for the input graph is computed.

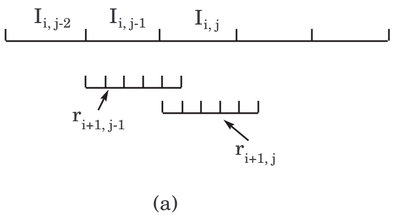
Because there are $O(\log\log n)$ processes, the processors used is $(m+n)\log\log n$. The algorithm can be run on the EREW PRAM. The only place where we need avoid concurrent read is to obtain hooking edges from the minimum spanning trees found so far for each level.

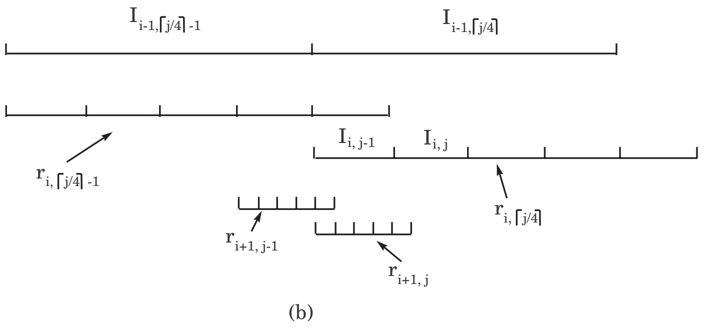
Level i consults the minimum spanning trees (obtaining the hooking edges) $c4^i$ times, where c is a constant. Thus the number of consults forms a geometric series. The bottom level consults the minimum spanning tree $c \log n$ times. Therefore if we allow constant time for one consult the total time for all consults will be $O(\log n)$.

Lemma 4: Property 1 is maintained in our algorithm. That is, edges used for hooking in an interval are in the active lists of the interval.

Proof: We consider two cases. The first case is when hooking happens in an effective interval. The second case is when hooking happens in an initialization interval.

Case 1: Consider an effective interval which is the j-th interval $I_{i,j}$ at level i. Consider j-th runs $r_{i+1,j-1}$ and $r_{i+1,j}$ at level $i+1$. It is easy to see from our algorithm that the edges used for hooking in interval $I_{i,j}$ are either in the active lists of the run $r_{i+1,j}$ or in the active lists of the effective intervals of run $r_{i+1,j-1}$. Both $I_{i,j}$ and $r_{i+1,j}$ obtain their active lists from the $(j-1)$-th interval at level i (i.e. interval $I_{i,j-1}$). Active lists in $I_{i,j}$ have larger size than corresponding active lists in $r_{i+1,j}$. If edges used for hooking are in active lists in $r_{i+1,j}$ then these hooking edges are also in the active lists of $I_{i,j}$. Now consider edges used for hooking in interval $I_{i,j}$ which are in the active lists in the effective intervals of run $r_{i+1,j-1}$. Let interval $I_{i,j-1}$ starts at stage s. Let both $I_{i,j-1}$ and $I_{i,j}$ be in run $r_{i,\lceil j/4 \rceil}$ at level i. At the end of $I_{i,j-1}$ active lists A_1 at level i are constructed and they are clean at stage s. At the end of initialization interval of run $r_{i+1,j-1}$ the active lists A_2 of run $r_{i+1,j-1}$ are also constructed and clean at stage s. Consider two situations.





$I_{i-1,\lceil j/4 \rceil-1}$
$I_{i-1,\lceil j/4 \rceil}$
$I_{i,j-1}$
$I_{i,j}$
$r_{i,\lceil j/4 \rceil-1}$
$r_{i,\lceil j/4 \rceil}$
$r_{i+1,j-1}$
$r_{i+1,j}$
(b)

Fig. 4. Illustration for Lemma 4.

Case 1.1. $I_{i,j-1}$ is an effective interval. See Fig. 4(a). Then both A_1 and A_2 obtain their edges from the active lists constructed at the end of interval $I_{i,j-2}$. At that moment active lists in A_1 have larger size than corresponding active lists in A_2. Therefore after active list construction active lists A_1 at the beginning of interval $I_{i,j}$ contains all edges in A_2 at the beginning of second interval of run $r_{i+1,j-1}$. Because run $r_{i+1,j}$ obtains its active lists A_3 from A_1 we conclude that edges in A_2 are also in A_3. That is, edges in the active lists in the effective intervals of run $r_{i+1,j-1}$ are all in the active list in the initialization interval of run $r_{i+1,j}$. Thus edges in the active lists in the effective intervals of run $r_{i+1,j-1}$ are also in the active lists of interval $I_{i,j}$. Therefore edges used for hooking in the interval $I_{i,j}$ are in the active lists of $I_{i,j}$.

Case 1.2. $I_{i,j-1}$ is an initialization interval of run $r_{i,\lceil j/4 \rceil}$. See Fig. 4(b). In this case run $r_{i,\lceil j/4 \rceil}$ corresponds to interval $I_{i-1,\lceil j/4 \rceil}$ at level $i-1$ and run $r_{i,\lceil j/4 \rceil -1}$ corresponds to interval $I_{i-1,\lceil j/4 \rceil -1}$ at level $i-1$. If $I_{i-1,\lceil j/4 \rceil -1}$ is an effective interval then by Case 1.1 we know that edges in the active lists for the effective intervals of run $r_{i,\lceil j/4 \rceil -1}$ are in the active lists for the initialization interval of run $r_{i,\lceil j/4 \rceil}$. From this we can conclude that the edges in the active lists of the effective intervals of run $r_{i+1,j-1}$ are in the active lists of the initialization interval of run $r_{i+1,j}$ and therefore they are also in the active lists of interval $I_{i,j}$. If $I_{i-1,\lceil j/4 \rceil -1}$ is an initialization interval we recurse to Case 1.2 but at a lower numbered level because we need to show now that the edges in the active lists of the effective intervals of run $r_{i,\lceil j/4 \rceil -1}$ are in the active lists of the initialization interval of run $r_{i,\lceil j/4 \rceil}$. Note that there is no initialization interval at level 1. Therefore the recursion eventually returns.

Case 2: Consider an initialization interval which is the j-th interval $I_{i,j}$ at level i. This proof of Case 1.2 can be used here to show that the edges in the active lists of the effective intervals of run $r_{i,\lceil j/4 \rceil}$ are in the active lists of the initialization interval of run $r_{i.\lceil j/4 \rceil}$, i.e., in the active lists of interval $I_{i,j}$. □

By the way we structured our algorithm, the threshold of active lists at lower numbered levels is no smaller than the threshold of corresponding active lists at higher numbered levels. Thus if Property $1'$ is maintained at lower numbered levels then by Lemma 4 it is also maintained at higher numbered levels. Property 2 is maintained at each level in our algorithm.

Theorem 1: There is an EREW minimum spanning tree algorithm with time complexity $O(\log n)$ using $(m+n)\log\log n$ processors (or $O((m+n)\log n \log\log n)$ operations). □

4. Reducing the Operation (Processor) Complexity

The main idea in this section is to use various techniques such as parallel integer sorting [2, 13, 14], parallel graph reduction [4], parallel selection [8] and parallel approximate compaction [12] within the minimum spanning tree algorithm given in Section 3. As a result we can achieve better operation complexity than previously known algorithms.

Parallel integer sorting is mainly used to eliminate internal and multiple edges. Each edge is represented by a pair of vertices and vertices can be represented by integers. Therefore each edge can be represented by a pair of integers. Thus by using integer sorting we can group multiple edges between two supervertices together and then remove these multiple edges. Internal edges can first be marked and then be eliminated by sorting them to the end of the array containing edges. Currently the best integer sorting algorithm on the EREW

PRAM sorts n integers in $\{0, 1, ..., m-1\}$ in time $O(\log n)$ with $O(n\sqrt{\log n})$ operations [13] using word length (the number of bits in a word) $\log(m+n)$. Note that the time and operations do not depend on m. On the CRCW PRAM the currently best result [2, 14] sorts n integers in $\{0, 1, ..., m-1\}$ in $O(\log n)$ time and $O(n \log\log n)$ operations using word length $\log(m+n)$. Again the time and operations do not depend on m. Note that we will also use a special property on the CRCW PRAM for sorting multiple edges. We can use concurrent write to identify whether an edge has multiples or not in constant time. If an edge does not have multiples then that edge does not need to participate in the sorting. Only edges having multiples need to participate in the sorting. In this way we can reduce the operation complexity.

Parallel graph reduction can be applied when we have a minimum spanning forest: if currently we have a forest of t trees then all the vertices in each tree can be combined into one vertex. As a result we will have only t vertices left and the maximum number of edges left will be t^2. When vertices in each tree is combined into one vertex the multiple edges between any two trees need to be eliminated. Again we can use integer sorting for eliminating multiple edges and this will involve nonlinear operations. For dense graphs with $\Omega(n^2)$ edges we can implement graph reduction in linear (i.e. $O(n^2)$) operations. This is done by renumbering the vertices such that vertices in one tree gets consecutive integer numbers. Edges are thus renumbered accordingly. Renumbering takes $O(n^2)$ operations with n^2 edges. After renumbering the multiple edges are adjacent in the $n \times n$ matrix A where $A[i, j]$ is the edge (i, j) and therefore they can be eliminated easily. Thus graph reduction takes linear operations for dense graphs and takes the same operations as integer sorting for sparse graphs.

Parallel selection is used for selecting minimum edges to construct active lists. Currently the best selection algorithm [8] on the EREW PRAM selects k-th item among n items in $O(\log n)$ time and $O(n \log^{(c)} n)$ operations or in $O(\log n \log^* n)$ time and $O(n)$ operations, where c is a constant. For our purpose k is always no larger than $n/\log n$. For this range of k we adapt the algorithm in [8] to obtain an EREW selection algorithm with $O(\log n)$ time and $O(n)$ operations.

The input to parallel approximate compaction is an array of 0's and 1's. If the array's size is n and there are k 1's then the currently best CRCW parallel approximate compaction algorithm [12] packs the k 1's to the first ck cells of the array, where c is a constant, in $O(\log\log n)$ time and $O(n)$ operations. Approximate compaction can be used to reallocate processors. The main advantage is the algorithm's fast time complexity. We can view initially that k tasks are allocated to the array with one task associated with each cell of the array valued with 1. Without compaction we would need n processor, one for each cell, for the k tasks. If each task takes t operations we would need nt operations. With approximate compaction we first pack the 1's to the beginning of the array and this takes $O(n)$ operations. We then execute the tasks which will then take $O(kt)$ operations. Thus we need only $O(n+kt)$ operations. Note that a simple prefix computation can accomplish the compaction except that it will take $O(\log n)$ time. In our situation we need a fast compaction algorithm. That is the reason we resort to approximate compaction which takes only $O(\log \log n)$ time.

4.1. An Algorithm with $O((m+n)\log n)$ Operations

We can force that each adjacency list contains at most m/n edges by splitting each vertex with an adjacency list of $a > m/n$ edges into $\lceil a/(m/n-2) \rceil$ vertices and distributing the edges evenly among the vertices and adding $\lceil a/(m/n-2) \rceil - 1$ edges with $-\infty$ weights between the vertices. It is easy to see that the new resulting graph has $O(n)$ vertices and $O(m)$ edges and the minimum spanning tree of the original graph can be easily obtained from the minimum spanning tree of the resulting graph.

Because each vertex has only m/n edges and an active list at lower numbered levels should have more than m/n edges we need not to build these lower numbered levels. For example if an active list at level i should have $(m/n)^2$ edges and an active list at level $i+1$ should have m/n edges and if we were to build active lists at both levels the active lists at both levels will be identical. That is the computation at level i is redundant to the computation at level $i+1$. For this reason there is no need to build levels numbered smaller than $i+1$. As computation proceeds active lists are combined to form larger active list and therefore at later stages we need to build lower numbered levels.

We use several phases to compute the minimum spanning trees. Because each adjacency list has only m/n edges, in the first phase we need only build levels $(1/2)\log \log n$ down to level i_1, where $2^{12\log n/4^{i_1}} = m/n$ or $i_1 = (1/2)\log \dfrac{12\log n}{\log(m/n)}$. If we were to build lower numbered levels, they would only duplicate the compressing process at level i_1. Because we build only to level i_1 and because the number of edges in the active lists at each level forms a geometric series , the number of processors used is $O(m+n)$. Also because we build only to level i_1, in the first phase we need to execute only 4 intervals of level i_1. However, after the first phase there will be only $n/2^{4\log n/4^{i_1}} = n/(m/n)^{1/3}$ supervertices and each supervertex has $(m/n)^{4/3}$ edges. We then execute the second phase. We rebuild each level. This time we build levels $(1/2)\log\log(n/(m/n)^{1/3})$ down to i_2, where $i_2 = (1/2)\log \dfrac{12\log(n/(m/n)^{1/3})}{\log(m/n)^{4/3}}$. Again because the number of edges at each level forms a geometric series, we need only $O(m+n)$ processors. We then execute 4 intervals at level i_2 in this phase. We then execute the third phase, and so on. In each next phase we build up to lower numbered levels. The number of processors used is always $O(m+n)$ in every phase. After we executed $\log n$ stages, the minimum spanning trees are computed. In each successive phases the levels are built bottom up. The initialization of active lists in each phase can be done as follows. Suppose we are to build from level i to level j. Then the active list at level $i-k$ has 2^{2^k} edges. Initially we have the active lists at level j. Each of these active list has $2^{2^{i-j}}$ edges. By using the parallel selection algorithm [8] (which takes $O(n\log^{(c)} n)$ operations and $O(\log n)$ time) to select $2^{2^{i-j-1}}$ smallest edges from each active list at level j we build the active lists at level $j+1$. Again using the selection algorithm [8] to select $2^{2^{i-j-2}}$ smallest edges from each active list at level $j+1$ we form the active lists at level $j+2$, and so on. The total operation spent is $O((m+n)\log^{(c)} n)$ (c is a constant which can be chosen as, say, 3 and then $\log^{(3)} n$ is $\log\log\log n$) and the time is $O(2^{i-j})$. Therefore we have:

Theorem 2: There is an EREW minimum spanning tree algorithm with time complexity $O(\log n)$ using $O((m+n)\log n)$ operations. $\square$

4.2. The Dense Graph Case

For dense graphs with $m = \Omega(n^2)$ we can obtain an optimal algorithm by using a selection algorithm.

Lemma 5: The selection of the element ranked $i \le n/\log n$ among n elements can be done in $O(\log n)$ time and $O(n)$ operations on the EREW PRAM.

Proof: Put n items into an array of $n/\log n$ rows and $\log n$ columns. For each row select the $(\lceil\sqrt{i/n}\log n\rceil)$-th smallest item using Cole's selection algorithm [8] which takes $O(\log\log n \log^* n) < O(\log n)$ time and $O(n)$ operations (or, as suggested by a referee, use sequential algorithm to select). The total number of selected items is $n/\log n$. Put these $n/\log n$ selected items in set B. Each item b of B corresponds to row b_r of the $n/\log n$ rows before selection, where b_r is the row containing b. Then select the $(\lceil\frac{n}{\log n}\sqrt{\frac{i}{n}}\rceil)$-th smallest items a among these $n/\log n$ items in B using [8] (here using the other version of the algorithm in [8] which takes $O((n/\log n)\log^{(c)} n) < O(n)$ operations and $O(\log n)$ time). By the way we select a, there are $\lceil\frac{n}{\log n}\sqrt{\frac{i}{n}}\rceil$ rows with $b \le a$. Because for each such row there are $\lceil\sqrt{i/n}\log n\rceil$ items $\le b$ there are at least $\frac{n}{\log n}\frac{i}{n}\log n = i$ items less than a. By the same reason there are at least $n - n/\log n - 2n/\sqrt{\log n}$ items greater than a (note that $i \le n/\log n$). Removing all items greater than a (all these eliminated items have rank $> i$) we have at most $n/\log n + 2n/\sqrt{\log n}$ items remaining. Put these remaining items in set C. Now we can select the i-th item in C, again using [8] (use the version with $O((n/\sqrt{\log n})\log^{(c)} n) < O(n)$ operations and $O(\log n)$ time). The time complexity of the whole selection process is $O(\log n)$ with $O(n)$ operations on the EREW PRAM. $\square$

Lemma 5 can be easily extended to select any element ranked $\le n/\log^{(k)} n$ for an arbitrarily large constant k.

We select $\log n$ minimum edges from the adjacency list of each vertex. We build levels $(1/2)\log\log n$ down to i_3, where $i_3 = (1/2)(\log\log n - \log\log\log n)$, using selected edges. We then execute 1 interval at level i_3. After that there will be $n/\log n$ supervertices left. We then apply the graph reduction technique. We renumber vertices such that vertices being hooked into one supervertex are numbered consecutively. Edges are renumbered according to the numbers given to the vertices. This renumbering enables us to move internal edges and multiple edges incident to a supervertex into consecutive memory locations and therefore they can be removed. Note that renumbering can be done by prefix computation and it takes only $O(\log n)$ time and $O(n^2)$ operations. We therefore avoid the sorting operation. After internal and multiple edges are removed there are $n/\log n$ supervertices and $O((n/\log n)^2)$ edges remaining. That is, we reduced the size of the graph from n vertices and $O(n^2)$ edges to $n/\log n$ vertices and $O((n/\log n)^2)$ edges. Now we can use Theorem 2 to compute the minimum spanning trees in additional $O(\log n)$ time and

$O((n/\log n)^2 \log n) = O(n^2/\log n)$ operations.

Theorem 3: There is an EREW minimum spanning tree algorithm with $O(\log n)$ time using $O(n^2)$ operations. □

When applied to dense graphs the algorithm in Theorem 3 is more efficient than the algorithm in Theorem 2. For dense graphs the algorithm in Theorem 3 is optimal because the number of operations used is $O(n^2)$ which is the lower bound.

4.3. Reducing Processor Complexity on the CRCW and EREW PRAM

We now explain how to reduce further the number of processors in Theorem 2. There are two places where we used $O((m+n)\log n)$ operations. The first is merging the active lists of several vertices into one active list of the supervertex after hooking. The second is the sorting used to sort internal and multiple edges together and used to select minimum edges in the active list.

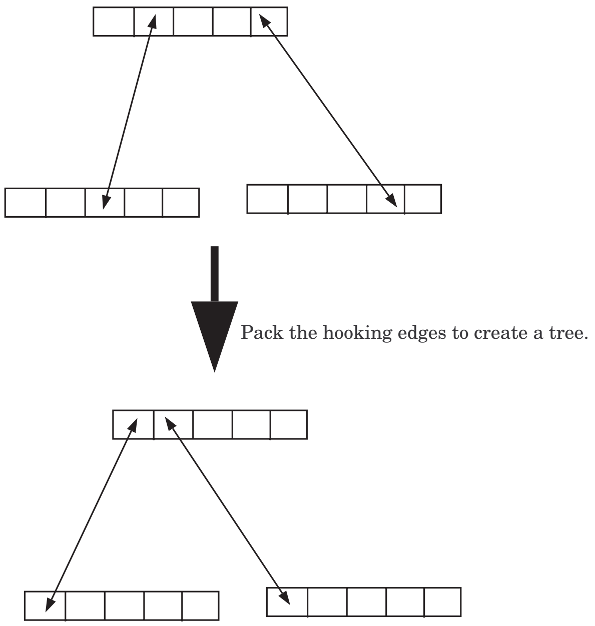


Fig. 5. Converting a tree of edges to a tree of vertices.

One way to merge the active lists is to use pointer jumping as this is used in [5, 16]. With m edges and doing pointing jumping for $\log n$ steps would need $O(m \log n)$ operations. We do not do pointer jumping directly. We store each active list in an array such that edges in an active list are stored in consecutive memory locations. In merging active lists, we first construct a tree T of vertices (instead of the active lists) representing the hooking from the hooking of the active lists. If each active list contains b edges and t vertices are hooked into a supervertex, the tree T we build has only those t vertices instead of tb edges. T can be obtained by a prefix computation on each active list. This is done by shrinking the active list of edges as shown in Fig. 5. If we were to do pointer jumping on a linked list of tb edges we would need $O(tb\log b)$ operations. What we do instead of pointer jumping on a linked list of tb edges is the pointer jumping on a linked list of t vertices. When we construct a b-active-list we need to do $O(\log b)$ steps of pointer jumping to combine the $O(b)$ active lists into one active list. When $t = O(b)$ active lists are hooked together to form a supervertex there are tb edges in the linked list of edges. Suppose we did pointer jumping on this linked list we would incur $O(tb\log b)$ operations. Now because we do pointer jumping on the linked list of t vertices the operations for pointer jumping become $O(t\log b)$. Plus $O(tb)$ operations needed to convert the linked list of edges to the linked list of vertices the total operations for merging active list become $O(tb + t\log b) = O(tb)$. The linked list of vertices is obtained from the Euler tour of T[20]. Therefore we have showed that the merging of active lists takes $O(\log n)$ time and $O(m+n)$ operations on the EREW PRAM for the whole algorithm.

Sorting is used to sort edges such that internal edges and multiple edges incident to two supervertices are moved into consecutive locations in memory for removal. Sorting is also used to find the minimum (say s) edges so that the active list can be constructed. The main idea here is to use integer sorting to remove internal and multiple edges and to use selection to select minimum edges to construct active lists.

Here we use integer sorting algorithms to sort internal and multiple edges. This is possible because each edge is represented by (a, b), where a and b are integers in the range $\{1, 2, ..., n\}$ representing vertices. On the CRCW PRAM n integers can be sorted in $O(\log n)$ time with $O(n\log\log n)$ operations [2, 14]. For a very sparse input graph, we use $O(\log\log n)$ phases. If the c-th phase builds down to level i, then there are $O(m)$ edges at level i at the beginning of the phase (the number of edges at higher numbered levels are geometrically decreasing and therefore is dominated by the edges at level i). We do not build levels numbered smaller than i as we explained in Section 4.1. To sort all these edges it takes $O(\log n/4^i)$ time (in one interval) and $O((m + n)\log\log n)$ operations. For $O(\log\log n)$ phases the time is $O(\log n)$ and the number of operations is $O((m + n)(\log\log n)^2)$.

We use the following method to further reduce the number of operations. In the sorting each integer incurs $O(\log\log n)$ operations [2, 14]. Before sorting we use concurrent write to determine for each edge e whether e is the unique edge incident to two supervertices s_1 and s_2. Here each edge e incident to supervertices s_1 and s_2 writes e into memory cell (s_1, s_2). When this concurrent write has no collision then e is the unique edge. Otherwise it is not the unique edge. If e is the unique edge then e does not participate in sorting and therefore e incurs only a constant number of operations for the sorting. If e is not the unique edge (there are other edges incident to s_1 and s_2), then e participates in the sorting and therefore e incurs $O(\log\log n)$ operations. Because multiple edges are removed (except for the minimum edge among the multiple) after sorting, each removed edge incurs $O(\log\log n)$ operations. We can associate the $O(\log\log n)$ operation incurred by the minimum edge among the multiple with a multiple edge which is removed after sorting and therefore the minimum edge now incurs only a constant number of operations. Thus if e is not removed it incurs a constant number of operations in one phase. For $O(\log\log n)$ phases e incurs $O(\log\log n)$ operations. Each edge can be removed only once, therefore each edge incurs only $O(\log\log n)$ operations and the total number of operations becomes $O((m + n)\log\log n)$.

Note that processors have to be reallocated because there are two sets of edges, a set S_1 of unique edges and a set S_2 of multiple edges and we have to reallocate processors associated with S_1 to work on S_2. We cannot use standard prefix computation to reallocate processors since reallocation needs $O(\log n)$ time. Note that we are reallocating about $m/\log n$ processors and during level i we can expend only $O(\log n/4^i)$ time. However, processor reallocation can be done in $O(\log\log n)$ time and $O(m+n)$ operations on the CRCW PRAM by the algorithm given in [12]. Thus for level $i < (1/2)(\log\log n - \log\log\log n)$ we can use integer sorting to remove internal and multiple edges because each interval at such level has at least $\Omega(\log\log n)$ stages while our integer sorting takes also $O(\log\log n)$ time. For level $i \geq (1/2)(\log\log n - \log\log\log n)$ we use comparison sorting [1, 7] and therefore our sorting takes only $t = O(\log n/4^i)$ time (which is what we wanted) and $O(mt) = O(m\log\log n)$ operations (which is OK for us).

On the EREW PRAM n integers can be sorted in $O(\log n)$ time with $O(n\sqrt{\log n})$ operations [13]. Therefore the operation complexity for removing internal and multiple edges becomes $O((m+n)\sqrt{\log n})$ while the time complexity is kept at $O(\log n)$. Here there is no need to distinguish between unique and nonunique edges because if phase p builds to level i and phase $p+1$ builds to level $j < i$, then the operations in these two phases is $O((m+n)\sqrt{\log n/4^i})$ and $O((m+n)\sqrt{\log n/4^j})$. Therefore the number of operations form a geometric series. The total number of operations become $O((m+n)\sqrt{\log n})$ even when unique edges participate in sorting at several levels.

Now we consider the sorting used for selecting minimum edges to construct active lists. We will use selection to reduce the number of operations. Suppose the lowest numbered level a phase is to build is i. Then there are $s = 2^{12\log n/4^i}$ edges in an active list at level i. Note that we do not need to worry about the edges at higher numbered levels because the total number of edges at higher numbered levels is only a fraction of the number of edges at level i. For the first interval of the first run we use selection to select the $s/\log s$ smallest edges using the selection algorithm we explained before and then sort the selected edges by their weights using comparison sorting. We then use selected edges to build levels $(1/2)\log\log n$ down to $i+1$ for the current phase. We may need to increase the number of phases for the whole algorithm but we have avoided sorting the whole active list. At the beginning of the second interval of the first run at each level in the current phase we do not need to construct the active lists because the computation at higher numbered levels has not proceeded to the stage for the need of an active list construction at level i. At the beginning of all other intervals of the first and other runs at level i we obtain the active list of each supervertex containing $2^{13\log n/4^i}$ edges. Note here that because the EREW selection algorithm we outlined above selects only the $(n/\log n)$-th smallest element among n elements we increased the size of the active list from $2^{12\log n/4^i}$ given before to $2^{13\log n/4^i}$ and therefore $2^{13\log n/4^i}/\log 2^{13\log n/4^i} > 2^{12\log n/4^i}$.

Note that we have to clean active list first before we select the minimum $2^{13\log n/4^i}$ edges because if we select edges first we might end up with many multiple edges among the edges selected. We first clean the active list for each supervertex and obtain an active list containing no more than $2^{13\log n/4^i}$ edges. We clean active lists by using integer sorting as we explained before. We then use the selection algorithm to select the minimum $s \leq 2^{12\log n/4^i}$ edges. Thus we are selecting the $(2^{12\log n/4^i})$-th smallest element among $2^{13\log n/4^i}$ elements and by Lemma 5 we can do it with linear operations. We then sort these selected edges by their weights using comparison sorting [1, 7]. Because we use selection and because we are sorting on a fraction (at most $1/\log s$) of the edges, the total number of operations for the selection and sorting becomes $O(m+n)$.

The ideas explained above give us the following theorem.

Theorem 4: There is an EREW minimum spanning tree algorithm with time complexity $O(\log n)$ with $O((m+n)\sqrt{\log n})$ operations. Also there is an CRCW minimum spanning tree algorithm with time complexity $O(\log n)$ with $O((m+n)\log\log n)$ operations. $\Box$

5. Conclusions

We presented several parallel algorithms for minimum spanning trees. By the application of parallel selection algorithm [8], parallel approximate compaction [12], parallel graph reduction [4] and parallel integer sorting [2, 13, 14] we could improve on the efficiency of parallel minimum spanning tree algorithms. Further research is needed to improve on the algorithms presented in this paper.

Acknowledgment

The authors wish to express their sincere gratitude to the referees for their extremely careful reviewing work and for their constructive comments and suggestions which improved this paper considerably.

References

[1] M. Ajtai, J. Komlós, E. Szemerédi. Sorting in $c\log n$ parallel steps. *Combinatorica*, 3, pp. 1-19(1983).

[2] A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in linear time? *Proc. 1995 Symposium on Theory of Computing*, 427-436(1995).

[3] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. on Computers*, C-36, 1258-1263, 1987.

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

[5] K. W. Chong. Finding minimum spanning trees on the EREW PRAM. *Proc. 1996 International Computer Symposium (ICS'96)*, Taiwan, 1996, pp. 7-14.

[6] K. W. Chong, Y. Han, T. W. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. *Proc. 1999 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), Baltimore, Maryland,* 225-234(January 1999).

[7] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(1988), pp. 770-785.

[8] R. Cole. An optimally efficient selection algorithm. *Information Processing Letters*, **26**, 295-299(1987/88).

[9] S. Cook, C. Dwork, R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous write. *SIAM J. Comput.*, Vol. 15, No. 1, Feb. 1986. pp. 87-97.

[10] R. Cole, P. N. Klein, R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling, *SPAA'96*, pp. 243-250.

[11] A. Gibbons, W. Rytter. Efficient Parallel Algorithms. Cambridge University Press, 1988.

[12] T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. *Proc. 3rd Israel Symposium on Theory and Computing Systems*, 220-228(1995).

[13] Y. Han, X. Shen. Parallel integer sort is more efficient than parallel comparison sorting on exclusive write PRAMs. *Proc. 1999 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), Baltimore, Maryland,* 419-428(January 1999).

[14] Y. Han, X. Shen. Conservative algorithms for parallel and sequential integer sorting. *Proc. 1995 International Computing and Combinatorics Conference, XiAn, China, Lecture Notes in Computer Science* **959**, 324-333(August, 1995).

[15] D. S. Hirshberg, A. K. Chandra and D. V. Sarwate. Computing connected components on parallel computers. *Comm. ACM*, 22(1979), pp. 461-464.

[16] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *J. of Algorithms*, 19, 383-401(1995).

[17] D. R. Karger. Random sampling in graph optimization problems. Ph.D. thesis. Department of Computer Science, Stanford University, 1995.

[18] C. K. Poon, V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. *Proc. 8th Annual International Symposium on Algorithms and Computation*, 1997, pp. 212-222.

[19] M. Snir. On parallel searching. *SIAM J. Comput.*, 14, 3(Aug. 1985), pp. 688-708.

[20] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14, pp. 862-874(1985).

[21] C.D. Zaroliagis. Simple and work-efficient parallel algorithms for the minimum spanning tree problem. *Parallel Processing Letters*, Vol. 7, No. 1, 25-37(1997).