Parallel Processing Letters,

SIMPLE AND WORK-EFFICIENT PARALLEL ALGORITHMS FOR THE MINIMUM SPANNING TREE PROBLEM*

CHRISTOS D. ZAROLIAGIS
Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany†

Received (received date)
Revised (revised date)
Communicated by

ABSTRACT

Two simple and work-efficient parallel algorithms for the minimum spanning tree problem are presented. Both algorithms perform $O(m \log n)$ work. The first algorithm runs in $O(\log^2 n)$ time on an EREW PRAM, while the second algorithm runs in $O(\log n)$ time on a COMMON CRCW PRAM.

Keywords: Minimum spanning tree, parallel random access machine.

1. Introduction

The minimum spanning tree problem is one of the most fundamental and intensively studied problems in network optimization with many theoretical and practical applications [1]. Given a connected n-vertex, m-edge undirected graph G with real edge weights, the *minimum spanning tree* (MST) problem is to find a spanning tree of total minimum weight among all spanning trees of G.

Algorithms for the MST problem have been developed as early as 1926. Three approaches have been mainly followed, known as Kruskal's, Sollin's and Prim's algorithms, respectively [1]. All of these algorithms are simple and easy to implement in $O(m \log n)$ time [1]. A faster version of Sollin's algorithm was given in [16] running in $O(m \log \log n)$ time. The best sequential algorithm (a variant of Prim's approach) runs in $O(m \log \beta(m,n))$ time [11], where $\beta(m,n) = \min\{i : \log^{(i)} n \leq m/n\}$. All of the above algorithms are deterministic and run on the (classical) unit-cost random access machine (RAM) model of computation, where the only operations allowed on the edge weights are binary comparisons. Better, linear-time algorithms are known if randomization is allowed [14], or if more powerful models of computation are used [10].

In this paper, we investigate the MST problem for the parallel random access machine (PRAM) model of parallel computation (the parallel version of the

*This work was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT).
†Email: zaro@mpi-sb.mpg.de.

unit-cost RAM model). A PRAM employs a number of processors which operate synchronously and communicate by reading from or writing to a global (shared) memory. The PRAM model has three variations depending on how simultaneous accesses to the same memory location by more than one processors are handled: EREW (exclusive-read exclusive-write), CREW (concurrent-read exclusive-write) and CRCW (concurrent-read concurrent-write) PRAM. The CRCW PRAM has also variants depending on the rule applied to resolve the write conflicts: the COMMON CRCW PRAM (the weakest variant) in which all processors writing to the same memory location have to write the same value which is stored into that location; the ARBITRARY CRCW PRAM in which among all processors writing to the same memory location, one succeeds arbitrarily; and the PRIORITY CRCW PRAM (the most powerful variant) in which among all processors writing to the same memory location, the processor with lowest priority succeeds. (For more on the PRAM and its variations, see [12].)

A primary goal in parallel computation is to design *work*-efficient algorithms, i.e., algorithms that solve a problem by performing a total number of operations (time $\times$ number of processors) which is smaller than the work performed by previous parallel algorithms and is close to the running time of the best known sequential algorithm for solving the same problem.

Previous results for the MST problem on the PRAM model are based on Sollin's approach and are as follows. In [4] an $O(\log^2 n)$-time, $O(n^2)$-work algorithm for the CREW PRAM is given (which is optimal for very dense graphs, i.e., with $\Omega(n^2)$ edges). For sparser graphs, there is a better algorithm [13] running in $O(\log^{3/2} n)$ time and performing $O(m \log^{3/2} n)$ work on an EREW PRAM. Two algorithms are known that run in $O(\log n)$ time, both on a PRIORITY CRCW PRAM: the first one [2] performs $O(m \log n)$ work and it is quite simple; the second one [8] performs $O(m \log\log\log n)$ work, but requires very elaborate techniques. It is worth mentioning that the result in [13] constitutes the first breakthrough w.r.t. time for the EREW PRAM model, because a direct simulation of any of the CRCW PRAM algorithms would require a slowdown of $\Omega(\log n)$ on an EREW PRAM (see e.g., [12]). All the above results are deterministic PRAM algorithms. A linear-work parallel algorithm is known only if randomization is allowed [7]; that algorithm runs on a (randomized) ARBITRARY CRCW PRAM.

In this paper, we present two simple and work-efficient parallel algorithms for the MST problem. Both algorithms are deterministic, perform $O(m \log n)$ work and are based on Sollin's approach. The first algorithm runs in $O(\log^2 n)$ time on an EREW PRAM, while the second one runs in $O(\log n)$ time on a COMMON CRCW PRAM. The EREW PRAM algorithm is not faster than that in [13], but it is more work-efficient. The COMMON CRCW PRAM algorithm is both faster and more work-efficient than the algorithms in [2,8], since direct simulations of these algorithms on a COMMON CRCW PRAM would either increase the time or the total work performed by a logarithmic factor. (A p-processor PRIORITY CRCW PRAM can be simulated by a p-processor COMMON CRCW PRAM with

slowdown $\Omega(\log p/\log\log p)$ [3,9,12]. On the other hand, a p-processor PRIORITY CRCW PRAM can be simulated by a $p\log p$-processor COMMON CRCW PRAM with slowdown $O(1)$ [5,12].)

We believe that the strength of our algorithms lies on two facts: (i) they give asymptotically better results w.r.t. previous approaches; and (ii) they are simple and consequently easy to implement (especially the first algorithm) in the sense that their implementation is based on fundamental, well-understood routines (e.g., prefix computations, list ranking, sorting) that are supposed to be present in any library of parallel combinatorial algorithms.

2. Preliminaries

Throughout the paper, let $G = (V, E)$ be a connected undirected graph, where $n = |V|$ and $m = |E|$. Let also $wt : E \rightarrow I\!R$ be a weight function on the edges of G. Without loss of generality, we assume that no two edges have the same weight. Otherwise, we consider the triple $\langle wt(e), u, v\rangle$ as the weight of the edge $e = (u, v)$ and compare edge weights using the lexicographic order. We assume that G is given in its adjacency list representation, i.e. the vertices are given in an array and every vertex v has an associated linked list $A(v)$ of its incident edges. Furthermore, since every edge (u, v) in $A(u)$ has a mate edge (v, u) in $A(v)$, there is a pointer $mate(u, v)$ which points to the location of (v, u) in $A(v)$.

Let $G' = (V', E')$ be a connected subgraph of G, where $V' \subseteq V$ and $E' \subseteq E$. We call the edges in E' *internal* edges of G', and the edges of G with only one endpoint in V' *external* edges of G'.

We now briefly review Sollin's algorithm. The algorithm performs a number of iterations. In each iteration it maintains a minimum spanning forest F, i.e. a collection of MSTs; for convenience we call these trees *components*. Initially F contains n (trivial) components, each one containing a single vertex. In the last iteration F becomes a single component which is the required MST. In every iteration the algorithm performs the following steps:

(1) *Selection.* For every component T_i in F, select its external edge (u, v) with minimum weight.

(2) *Merging.* For every such edge (u, v), where $u \in T_i$ and $v \in T_j$ $(i \neq j)$, merge T_i and T_j into a single tree.

The correctness of the algorithm is based on the following well-known property (for a proof see e.g. [1]).

Lemma 1 *Let F be a minimum spanning forest of a graph G and let (u, v) be the external edge of minimum weight of a tree in F. Then, a minimum spanning tree of G contains the edge (u, v) and all edges in F.*

It is easy to verify that each iteration in Sollin's algorithm takes $O(m)$ time. To bound the number of iterations, observe that every iteration reduces the number of trees by a factor of at least $1/2$ (every tree is merged with another). Consequently,

the total number of iterations is $O(\log n)$ and thus the running time of Sollin's algorithm is $O(m \log n)$.

3. The EREW PRAM algorithm

The algorithm in this section is a natural parallelization of Sollin's approach. We will describe how the selection and merging steps are implemented efficiently on an EREW PRAM, using only basic routines: prefix computations, list ranking and the Euler-tour technique. All of them can be done in $O(\log p)$ time and $O(p)$ work on an EREW PRAM [12, Chap. 2 and 3], where p is the size of the input. In the following, we shall frequently perform prefix computations on lists rather than arrays, i.e. we will assume that the lists are already ranked. This is not a problem, since list ranking needs the same resource bounds as prefix computation.

Let F_i denote the minimum spanning forest in iteration i, and let T_j^i be the j-th component of F_i. With every such component T_j^i, we maintain a list V_j^i of its vertices and a list $Ext(T_j^i)$ of its external edges (where the last element is a null one). For the latter list, we maintain five pointers with every $e = (x, y) \in Ext(T_j^i)$: $first(e)$, $last(e)$, $prev(e)$ and $next(e)$ which point to the first, last (non-null), previous and next element, respectively, in $Ext(T_j^i)$, and $mate(e)$ which points to the mate of e, $\overline{e} = (y, x) \in Ext(T_k^i)$, in some other list $Ext(T_k^i)$. Every T_j^i is *represented* by its root node $r(T_j^i)$. During merging, trees of representatives are created (denoting implicitly the mergings of the components). For this reason, we maintain a pointer $P(r(T_j^i))$ pointing to the parent of $r(T_j^i)$ in the tree of representatives. All vertices $z \in V_j^i$ have a label $\ell(z) = r(T_j^i)$. Each T_j^i enters iteration i equipped with these data structures. We will show how the selection and merging steps during iteration i can be implemented using these data structures, and also how the data structures for the new components at the beginning of iteration $i+1$ are updated efficiently.

Algorithm Parallel-MST-1.

1. **for all** $v \in V$ **do in parallel**
   $V_v^1 = \{v\}$; $Ext(T_v^1) = A(v)$; $r(T_v^1) = v$; $P(r(T_v^1)) = r(T_v^1)$;
   od
2. **for** $1 \leq i \leq \lceil \log n \rceil$ **do** ITERATION(i);

End of algorithm.

The i-th iteration is implemented as follows.

Procedure ITERATION(i).

(A) (* Selection *)
For each component T_j^i do in parallel: select the external edge (u, v) $(u \in T_j^i)$ with minimum weight from $Ext(T_j^i)$, using the well-known balanced binary tree construction [12, Chap. 2] in the elements of $Ext(T_j^i)$.

(B) (* Merging *)
For all edges (u, v) $(u \in T_j^i,\ v \in T_k^i,\ k \neq j)$ selected in the previous step do in parallel:

(B1) If $r(T_j^i) > r(T_k^i)$, then merge T_k^i into T_j^i.

(B2) If $r(T_j^i) < r(T_k^i)$ and $P(r(T_j^i)) = r(T_j^i)$, then merge T_j^i into T_k^i.

Merging of T_k^i into T_j^i is done in two steps:
(M1) The edge list $Ext(T_k^i)$ is plugged into $Ext(T_j^i)$, by setting $next(last(mate(u,v))) = next(u,v)$; $prev(next(u,v)) = last(mate(u,v))$; $next(u,v) = first(mate(u,v))$; and $prev(first(mate(u,v))) = (u,v)$.
(M2) Create an edge from $r(T_k^i)$ to $r(T_j^i)$, by setting $P(r(T_k^i)) = r(T_j^i)$.
(Merging of T_j^i into T_k^i is done analogously.)

(C) (* Clean up *)

(C1) As a result of Step (B) several mergings between the various components in F_i may have occurred. These mergings are implicitly represented by the trees of representatives (created during Step (M2) of merging). Every node $r(T_q^i)$ in such a tree R_s corresponds to a component in F_i and the directed edge $(r(T_q^i), r(T_t^i))$ denotes that component T_q^i was merged into component T_t^i. In every R_s do the following. Let $r(T_p^i)$ be the root of R_s. Using the Euler-tour technique (see below), generate for every node $r(T_q^i)$ in R_s a pointer to $r(T_p^i)$. Then, for every $z \in V_q^i$ $(p \neq q)$, where $r(T_q^i) \in R_s$, set $\ell(z) = r(T_p^i)$.

(C2) If there is an edge (x,y) in some $Ext(\cdot)$ list such that $\ell(x) = \ell(y)$, then mark (x,y). Remove all marked edges from all $Ext(\cdot)$ lists and update the *first*, *last*, *next*, *prev* and *mate* pointers using a prefix computation. The resulting list is the list of external edges of the new component (that will enter iteration $i+1$) and is represented by the root of R_s.

(C3) Construct a preorder traversal $v_1, v_2, \cdots, v_{|R_s|}$ of the nodes in R_s and form a list. Then, replace in this list each node v_l $(1 \leq l \leq |R_s|)$, corresponding to some component T_k^i, by V_k^i. The resulting list is the vertex list of the new component represented by the root of R_s.

End of procedure.

An illustration on how the algorithm works is given in Figures 2 through 4, having as input the graph of Figure 1.

It is clear by Lemma 1 that all edges (u,v) selected in Step (A) form the required MST. Steps (B1) and (B2) ensure that (not only every component is merged with another one, but also) ties are broken in the case where both components T_j^i and T_k^i have selected the same edge (u,v), thus avoiding the creation of directed cycles of length 2 in the trees of representatives.

Before proceeding to the resource bounds, we discuss some implementation details regarding Steps (C1), (C2) and (C3) of the procedure. To apply the Euler-tour technique in Step (C1), it is required that: (1) every tree edge $(r(T_q^i), r(T_t^i))$ has a pointer to its anti-parallel edge $(r(T_t^i), r(T_q^i))$; and (2) the children of a node $r(T_t^i)$ in R_s are connected in a linked list. The former can be easily satisfied during Step

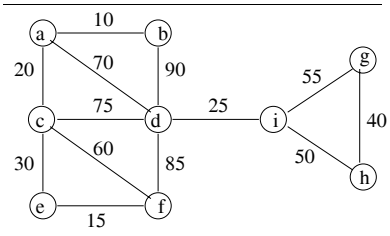

Fig. 1. Input graph with edge weights.

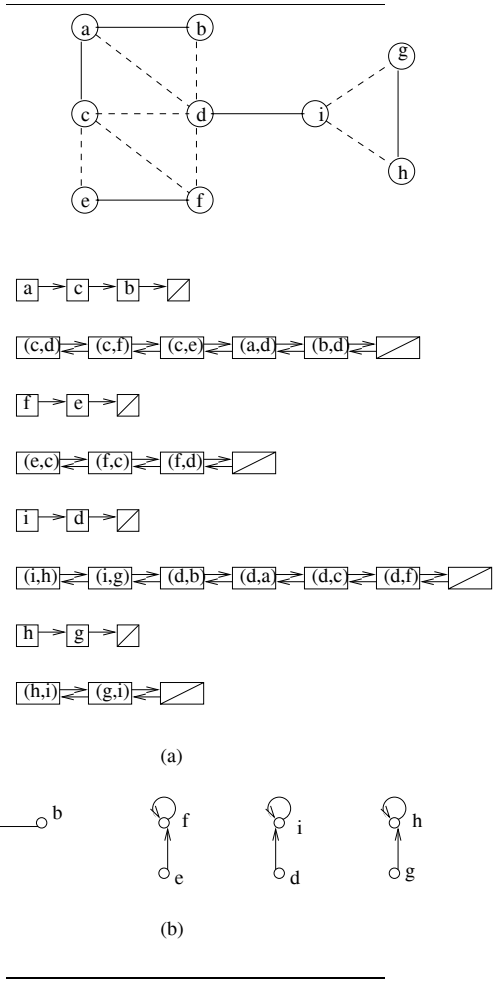

Fig. 2. (a) The data structures in the beginning of the 2nd iteration of Algorithm Parallel-MST-1. (b) The trees of representatives; arrows denote parent pointers.

(M2) of merging. The latter can be achieved in $O(\log n)$ time and $O(m)$ work, over all trees of representatives, as follows. After the execution of Step (B), mark in $Ext(T_t^i)$ the edge (u, v) that caused the merging of T_q^i into T_t^i. Create a linked list of all marked entries, using a prefix computation. Having this list, it is easy to generate the required linked list of the children of $r(T_t^i)$ in R_s. In Step (C2), the updating of $first$, $last$, $next$ and $prev$ pointers can be easily done with a prefix computation. The updating of $mate$ pointers is done as follows: Before removing the marked edges, compute for every unmarked edge (w, z) its new position in the $Ext(\cdot)$ list to which it belongs (by simply setting 0 for all marked entries, 1 for the unmarked and performing a prefix summation). Then, pass this information to a field in (z, w), say, $newmate(z, w)$. After removal of the marked edges, set $mate(w, z) = newmate(w, z)$. Finally, for Step (C3) note that having constructed an Euler-tour in every R_s, the required preorder traversal can be easily computed from that (see [12, Chap. 3] for the details).

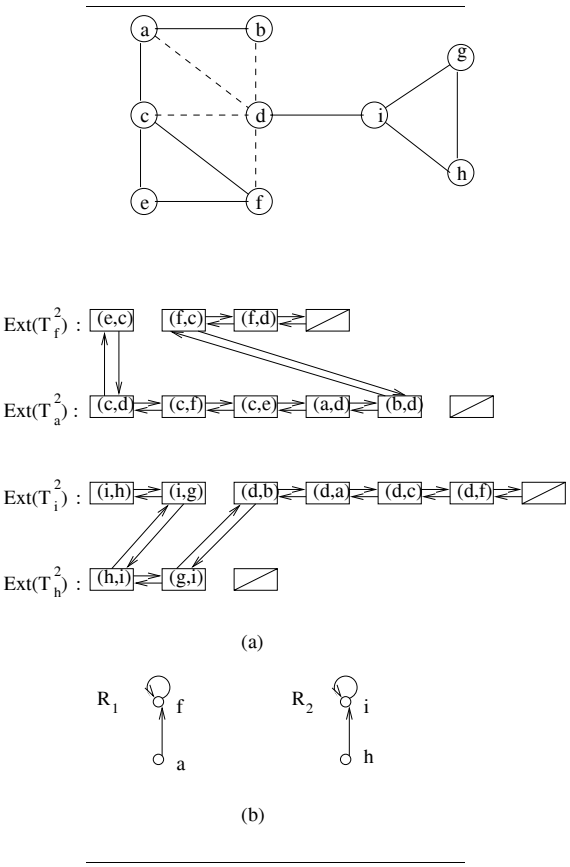

Fig. 3. (a) Merging of the components by appropriate rearrangement of pointers in the lists of external edges. (b) The new trees of representatives R_1 and R_2.

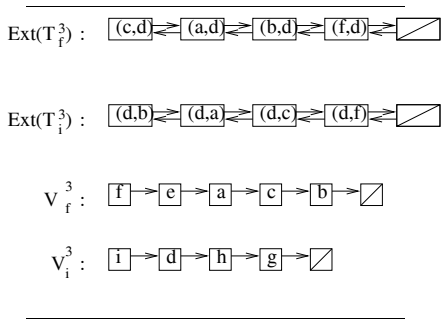

Fig. 4. The lists after the clean up phase, in the form required to enter the 3rd iteration of Algorithm Parallel-MST-1. The marked elements have been removed from the lists of external edges and there is a new list of vertices for each component. Note that the elements in V_f^3 consists of the elements of lists V_f^2 and V_a^2 which replace the elements f and a respectively, in the list resulted from a preorder traversal of R_1. An analogous observation holds for V_i^3.

Concerning the resource bounds, it is not hard to verify that Step (B) takes $O(1)$ time using $O(m)$ work, while all other steps need $O(\log n)$ time using $O(m)$ work. Hence, we have established the following.

Theorem 2 *Algorithm Parallel-MST-1 solves the minimum spanning tree problem in $O(\log^2 n)$ time performing $O(m \log n)$ work on an EREW PRAM.*

4. The COMMON CRCW PRAM algorithm

The algorithm in this section is based on the Awerbuch-Shiloach algorithm [2]. That algorithm constitutes a different implementation of Sollin's approach. Every component is a rooted tree and a rooted tree of depth 1 is called a *star*. The algorithm works as follows. A processor is associated with every edge (u, v) of G and performs repeatedly three steps until a certain condition is satisfied after which it becomes inactive. The algorithm terminates when every processor has become inactive. During the execution of the algorithm, a pointer $P(v)$ is maintained, for every vertex $v \in G$, pointing to the parent of v in the current component containing v. (Initially, $P(v) = v$, i.e. every vertex is a star.) Every active processor associated with the edge (u, v) executes the following steps:

(1) If (u, v) is an external edge of a star, then its associated processor tries to hook it into another component. Among the potentially many processors (associated with the external edges of the same star) attempting to do it, the succeeding one is the processor associated with the external edge of minimum weight. By Lemma 1, this edge belongs to the MST.

(2) As a result of Step (1), directed cycles of length 2 may be created. This can happen if (u, v) is an edge such that u belongs to a star S_1 and v belongs to a star S_2, and it is the minimum-weighted external edge for both S_1 and S_2. Then, in this step these directed cycles are detected and broken in favor of the vertex with smaller

number (simply by deleting the pointer of the smaller numbered vertex pointing to the larger numbered one).

(3) The components which are not stars reduce their depth by a factor of 1/3 at least. This is done by "shortcutting": if u does not belong to a star, then $P(u) = P(P(u))$; otherwise, the processor associated with (u, v) becomes inactive.

It is easy to verify that Step (1) can be done in $O(1)$ time, if the PRIORITY CRCW PRAM is used. The other two steps take $O(1)$ time, even in the COMMON CRCW PRAM.

Assume for the moment that $m \geq n \log^2 n$. (At the end of the section, we will show how to overcome this assumption.)

Our algorithm follows the above scheme, but manages to avoid the "expensive" hooking step by choosing a subset of the edges to participate in hooking. This is achieved by partitioning the adjacency list $A(u)$ of a vertex $u \in G$ in a manner similar to that in [16]: every $A(u)$ is partitioned into $\lceil \log n \rceil$ groups each of size $\lceil \frac{|A(u)|}{\log n} \rceil$ and such that for any two groups $A^i(u)$ and $A^j(u)$ with $i < j$, $wt(e) < wt(e')$ for any $e \in A^i(u)$ and $e' \in A^j(u)$ (see Figure 5.(a)). At every iteration a star S hooks using the external edges incident to every $u \in S$ that belong to only one such group of $A(u)$ (see Figure 5.(b)). If, during the execution of the algorithm, a group does not have external edges, then the next group containing external edges is considered. The details of our algorithm are given in Procedure MST-2 below. $T(e)$ is a Boolean variable associated with every edge $e \in G$; if $T(e) = 1$, then e belongs to the MST. Variable $W(u)$ contains the external edge of minimum weight whose associated processor is the succeeding one in hooking the star rooted at u into another component. The Boolean array B_u, associated with every $u \in G$, is used to determine the next group of $A(u)$ that contains at least one external edge (see Figure 5.(a)).

Procedure MST-2.

01. **for all** $u \in G$ **do in parallel**
    Partition $A(u)$ into $\lceil \log n \rceil$ groups, $A^{i_u}(u)$, each of size $\lceil \frac{|A(u)|}{\log n} \rceil$ such that $wt(e) < wt(e')$ for every $e \in A^{i_u}(u)$, $e' \in A^{j_u}(u)$ and $i_u < j_u$;
    od
02. **for all** $u \in G$ **do in parallel**
    Unmark all edges (u, v) in all groups of $A(u)$, and set $T(u, v) = 0$;
    Initialize array $B_u[1 : \lceil \log n \rceil]$;
    $i_u = 1$; $P(u) = u$; $W(u) =$ null;
    od
03. **while** $\exists$ unmarked edges **do**
    for all $u \in G$ **do in parallel**
04. **for all** unmarked (u, v) in $A^{i_u}(u)$ **do in parallel**
    (* Hooking *)
    if u belongs to a star **then**
    if $P(u) \neq P(v)$ **then**
    $P(P(u)) = P(v)$; $W(P(u)) = (u, v)$;

if $W(P(u)) = (u, v)$ **then** $T(u, v) = 1$;
else mark (u, v);
(* Breaking cycles *)
if $u < P(u)$ and $u = P(P(u))$ **then** $P(u) = u$;
od
05. (* Shortcut *)
for all unmarked (u, v) in $A(u)$ **do in parallel**
if u does not belong to a star **then** $P(u) = P(P(u))$;
od
06. **for all** (u, v) in $A^{k_u}(u)$, $i_u < k_u \leq \lceil \log n \rceil$ **do in parallel**
if u belongs to a star and $P(u) = P(v)$ **then** mark (u, v);
od
07. (* Choosing the next group *)
08. **for** every group $A^{k_u}(u)$, $i_u \leq k_u \leq \lceil \log n \rceil$ **do in parallel**
Perform an OR computation with marked entries as 0's
and unmarked entries as 1's, and store the result in $B_u(k_u)$;
od
09. **if** not all entries of B_u are 0 **then**
10. Determine the index j_u of the leftmost 1 in B_u;
$i_u = j_u$;
od
od
End of procedure.

The correctness of the procedure follows by Lemma 1 and the fact that in every iteration every star hooks, since for every vertex u belonging to a star, a group of its incident edges is considered that contains at least one external edge.

Concerning the resource bounds, first notice that the total number of iterations of the while-loop at Step 3 is $O(\log n)$. The proof is similar to that in [2] and is based on the following observations: (a) the sum of the depths of all components does not increase after an iteration; (b) all stars hook at every iteration; and (c) the components that are not stars reduce their depth by a factor of 1/3 due to Step 5, and consequently the sum of the depths of all components is reduced by the same factor.

Checking whether a vertex u belongs to a star can be easily done in $O(1)$ time [2], [12, Chap. 5]. Step 1 can be implemented in $O(\log n)$ time and $O(m \log n)$ work, using sorting [6]. Step 2 clearly needs $O(1)$ time and $O(m)$ work. The resource bounds of Step 4 are dominated by the execution of the hooking step (as explained before). Using the result in [5], this step takes $O(1)$ time and $O(\sum_{u \in G}(\frac{|A(u)|}{\log n}) \log(\sum_{u \in G}(\frac{|A(u)|}{\log n}))) = O(m)$ work on a COMMON CRCW PRAM. Steps 5 and 6 take clearly $O(1)$ time and $O(m)$ work. Step 8 needs $O(1)$ time and $O(\sum_{u \in G} |A(u)|) = O(m)$ work. The "if-condition" in Step 9 can be checked within the same resource bounds. Step 10 can be implemented using the Find-First algorithm in [5], or the Leftmost-Prisoner algorithm in [12, Chap. 10], and therefore

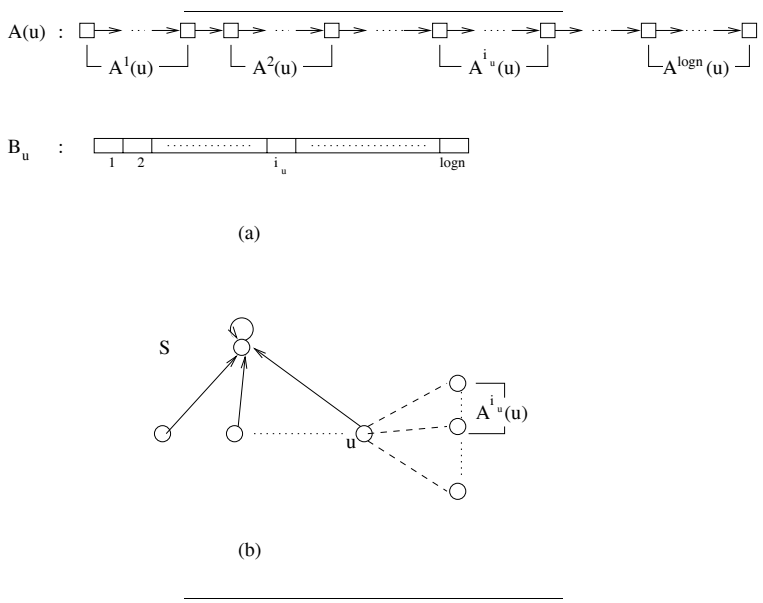

Fig. 5. (a) The adjacency list of a vertex u of the graph and its associated Boolean array B_u. (b) During the execution of Algorithm Parallel-MST-2, a star S hooks using those adjacent vertices of a vertex $u \in S$ that belong to only one group $A^{i_u}(u)$.

needs $O(1)$ time and $O(n \log n \log(n \log n)) = O(m)$ work. Hence, Step 7 can be performed in $O(1)$ time and $O(m)$ work. Consequently, Procedure MST-2 runs in $O(\log n)$ time performing $O(m \log n)$ work on a COMMON CRCW PRAM.

If $m < n \log^2 n$, we first reduce the number of vertices in G (by executing a few iterations of the Awerbuch-Shiloach algorithm) and then perform Procedure MST-2 on the reduced graph. The complete algorithm presented in this section is as follows.

Algorithm Parallel-MST-2.
if $m \geq n \log^2 n$ **then** Run Procedure MST-2 on G
else

(1) Run the Awerbuch-Shiloach algorithm in G – using the simulation procedure for the COMMON CRCW PRAM with $O(\log n / \log \log n)$ slowdown [3,9], [12, Chap. 10] – until every component C (star or nonstar) has at least $\log^2 n$ vertices. Since at every iteration the size of the smallest component at least doubles, $2\lceil \log \log n \rceil$ iterations of the Awerbuch-Shiloach algorithm suffices.

(2) For every component C do: remove the internal edges of C and shrink C into a single vertex $r(C)$, using pointer jumping. Now, all the external edges of C are incident to $r(C)$. Call the resulting graph $G' = (V', E')$. Note that $|V'| \leq n / \log^2 n$ and $|E'| \leq m$.

(3) Run Procedure MST-2 in G'.

End of algorithm.

It is easy to verify that each one of the three steps in the "else-part" needs $O(\log n)$ time and $O(m \log n)$ work. Hence, we have established the following.

Theorem 3 *Algorithm Parallel-MST-2 solves the minimum spanning tree problem in $O(\log n)$ time performing $O(m \log n)$ work on a* COMMON *CRCW PRAM.*

It is worth noting that the only routines needed for the implementation of Algorithm Parallel-MST-2 are sorting, prefix computations and the simulation procedures for simulating a PRIORITY CRCW PRAM on a COMMON CRCW PRAM. Both simulations are in turn based on the Leftmost-Prisoner algorithm whose implementation is simple and described in [12, p.499]).

5. Closing remarks

We have presented two simple and work-efficient parallel algorithms for the MST problem. We believe that their simplicity makes the algorithms easy to implement, since their implementation is based on fundamental, well-understood routines that are likely to be found on any library of parallel combinatorial algorithms. A possible environment for such an implementation is the PAD library of basic PRAM algorithms and data structures [15], currently under development.

Acknowledgements

I am grateful to the anonymous referees for their comments which improved the presentation of the paper. I would also like to thank Jesper Träff for many helpful discussions and comments on an earlier version of this paper.

References

[1] R. Ahuja, T. Magnanti and J. Orlin, *Network Flows* (Prentice-Hall, 1993).
[2] B. Awerbuch and Y. Shiloach, New Connectivity and MSF algorithms for Shuffle-Exchange Network and PRAM, *IEEE Trans. on Computers* **36:10** (1987) 1258-1263.
[3] R. Boppana, Optimal separations between concurrent-write parallel machines, in *Proc. 20th ACM Symp. on Theory of Computing* (ACM, 1989) 320-326.
[4] F. Y. Chin, J. Lam and I.N. Chen, Efficient parallel algorithms for some graph problems, *Communications of the ACM* **25:9** (1982) 659-665.
[5] B. Chlebus, K. Diks, T. Hagerup and T. Radzik, New Simulations between CRCW PRAMs, in *Fundamentals of Computation Theory* – FCT'89, *Lecture Notes in Computer Science* **380** (Springer-Verlag, 1989) 95-104.
[6] R. Cole, Parallel Merge Sort, *SIAM J. on Computing* **17:4** (1988) 770-785.
[7] R. Cole, P.N. Klein and R.E. Tarjan, A linear-work parallel algorithm for finding minimum spanning trees, in *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures* (ACM, 1994) 11-15.

[8] R. Cole and U. Vishkin, Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems, in *Proc. 27th IEEE Symp. on Foundations of Comp. Sc.* (IEEE, 1986) 478-491.

[9] F. Fich, P. Ragde and A. Wigderson, Simulations among concurrent-write PRAMs, *Algorithmica* **3:1** (1988) 43-51.

[10] M. Fredman and D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, in *Proc. 31st IEEE Symp. on Foundations of Comp. Sc.* (IEEE, 1990) 719-725.

[11] H. Gabow, Z. Galil, T. Spencer and R.E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica* **6:2** (1986) 109-122.

[12] J. JáJá, *An Introduction to Parallel Algorithms* (Addison-Wesley, 1992).

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

[14] D.R. Karger, P.N. Klein and R.E. Tarjan, A randomized linear-time algorithm to find minimum spanning trees, *Journal of the ACM* **42:2** (1995) 321-328.

[15] C.W. Kessler and J. L. Träff, A library of basic PRAM algorithms and its implementation in FORK, in *Proc. 8th Symp. on Parallel Algorithms and Architectures (SPAA'96)* (ACM, 1996) 193-195.

[16] A.C. Yao, An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees, *Information Processing Lett.* **4:1** (1975) 21-23.