Information Processing Letters 27 (1988) 227–23
North-Holland 28 April 1988

PARALLEL ALGORITHMS FOR COMPUTING MAXIMAL INDEPENDENT SETS IN TREES AND FOR UPDATING MINIMUM SPANNING TREES *

Hermann JUNG

Department of Mathematics, Humboldt University at Berlin, DDR-1086 Berlin, German Democratic Republic

Kurt MEHLHORN

Department of Computer Science, University of Saarland, D-6600 Saarbrücken, Fed. Rep. Germany

Communicated by T. Lengauer
Received 17 March 1987
Revised 27 October 1987

We apply the Cole–Vishkin technique of deterministic coin tossing to the computation of independent sets in trees and graphs of bounded degree. We also show how to update minimum spanning trees efficiently in parallel.

Keywords: Parallel algorithm, independent set, tree, spanning tree

1. Introduction

The model of computation used in this paper is the parallel random access machine (PRAM) in its various forms: concurrent-read, concurrent-write (CRCW), concurrent-read, exclusive-write (CREW), exclusive-read, exclusive-write (EREW). We study the problem of computing maximal independent sets in trees and graphs of bounded degree and the problem of updating minimum spanning trees. More specifically, our results are as follows.

Let T be an almost-tree (in an almost-tree, every vertex has outdegree at most one) on the vertex set $\{1, 2, \ldots, n\}$ given by the array of parent pointers. A set I of vertices of T is a *maximal dividing set* if I is a maximal independent set and removal of I dissects the tree T into trees of depth O(1). Fig. 1 shows a tree, a maximal independent set, and a maximal dividing set.

Theorem A. *A maximal dividing set in an almost-tree can be computed on a* CREW-PRAM *in* $O(\log^* n)$ *time using n processors, and in* $O(\log n)$ *time using* $n/\log n$ *processors.*

Cole and Vishkin [2] proved this result previously (for trees instead of almost-trees) in the stronger CRCW-model using their algorithm [1] for computing independent sets in linear lists as a subroutine. We observe on the contrary that the algorithm of [1] directly generalizes to almost-trees, thus simplifying and generalizing their result.

Theorem A was proved independently by Goldberg, Plotkin and Shannon [4]. Their algorithm is simpler to describe and to understand than ours.

In Theorem B we turn to maximal independent sets in graphs of bounded degree.

* This work was supported by the DFG, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität.

0020-0190/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

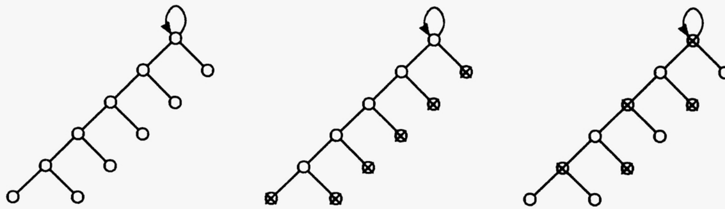

Fig. 1. A tree, a maximal independent set, and a maximal dividing set.

Theorem B. *A maximal independent set in a graph of bounded degree can be computed on an* EREW-PRAM *in* $O(\log^* n)$ *time using n processors and in* $O(\log n)$ *time using* $n/\log n$ *processors.*

A related result was previously shown by Hagerup, Chrobak and Diks [5]. They computed an independent set of size at least $n/2^d$ in $O(\log^* n)$ time using the EREW-PRAM. We prove Theorem B by repeated application of Theorem A. A maximal independent set in a general graph of n vertices and m edges can be found in $O(\log^2 n)$ time using $O(n^2 m)$ processors; this was shown by Luby [6].

Theorem B was also proved independently by Goldberg, Plotkin and Shannon [4].

In the second part of the paper we treat the problem of updating a minimum spanning tree upon the addition of a vertex. Let $T = (V, A)$ be a minimum spanning tree of the weighted graph $G = (V, V \times V, c)$; here, $c: V \times V \to \mathbb{R}$ is a weight function. Let $w \notin V$ be a new vertex, let $V' = V \cup \{w\}$, and let $c': V' \times V' \to \mathbb{R}$ be an extension of c. The problem is to compute a minimum spanning tree of the graph $G' = (V', V' \times V', c')$. Known algorithms for this problem operate in $O(\log n)$ time using $O(n^2)$ [7] and $O(n)$ [9] processors, respectively. We show the following theorem.

Theorem C. *Minimum spanning trees can be updated on a* CRCW-PRAM *in* $O(\log n)$ *time using* $n/\log n$ *processors.*

We prove Theorem C by showing that updating a minimum spanning tree is strongly related to the problem of evaluating a certain expression involving the operations min and max. The expression is given by the spanning tree to be updated with every vertex replaced by a small formula of min′ s and max′ s. The time and processor bounds then follow from a result of Cole and Vishkin [2].

2. Independent sets

In this section we show how to compute maximal dividing sets in almost-trees and maximal independent sets in graphs of bounded degree.

2.1. Definition. Let $G = (V, E)$ be a digraph.

(a) G is an almost-tree if the outdegree of every vertex of G is at most one.

(b) A set $I \subseteq V$ is a maximal dividing set of G if:

(b1) I is an independent set, i.e., for $v, w \in I$, $v \neq w$ implies $(v, w) \notin E$,

(b2) every path in $G - I$ has length at most one,

(b3) I is maximal with respect to set inclusion among the sets having properties (b1) and (b2).

We assume that almost-trees are given as integer array *suc*[1..*n*] and Boolean array *leaf*[1..*n*]. If *suc*[*i*] = 0, then *i* has outdegree 0 and if *suc*[*i*] ≠ 0, then (*i*, *suc*[*i*]) is the unique edge leaving *i*. Also, *leaf*[*i*] = *true* iff the indegree of *i* is *zero*.

Theorem A. *Let $T = (V, E)$ be an almost-tree with $n = |V|$. A maximal dividing set of T can be computed on a* CREW-PRAM *in* $O(\log^* n)$ *time using n processors or in* $O(\log n)$ *time using* $n/\log n$ *processors.*

Proof (The proof given here is the result of joint work with Torben Hagerup). Our proof is an extension of the proof by Cole and Vishkin for the corresponding result for linear lists. The main complication stems from the fact that vertices in almost-trees may have arbitrarily large indegree and that, therefore, a vertex cannot obtain information from its sons in the CREW model of computation. We start by reviewing an important fact on the ruling set problem from [1, Section 2.1].

2.2. Fact. *Let N be any positive integer and let $x_0, x_1, \ldots$ be any (finite or infinite) sequence with $x_i \in [0..N-1]$ and $x_i \neq x_{i+1}$ for all i. Then there are functions $f: [0..N-1]^{10} \to \{true, false\}$ and $y: [0..N-1]^2 \to [0..\lceil \log N \rceil - 1]$ such that the following hold for the sequences $s_0, s_1, s_2, \ldots$ and $y_0, y_1, y_2, \ldots$ where $s_i = f(x_i, x_{i+1}, \ldots, x_{i+9})$ and $y_i = y(x_{i+4}, x_{i+5})$. (Of course, s_i and y_i exist only if x_{i+9} and x_{i+5} respectively exist.)*

(a) s_i *implies* $\neg s_{i+1}$ *for all i.*
(b) $s_i \vee s_{i+1} \vee \cdots \vee s_{i+\lceil \log N \rceil}$ *for all i.*
(c) $\neg(s_i \vee \cdots \vee s_{i+3})$ *implies* $y_{i+1} \neq y_{i+2}$.
(d) $\neg(s_i \vee \cdots \vee s_{i+4})$ *implies* $y_{i+1} < y_{i+2} < y_{i+3}$ *or* $y_{i+1} > y_{i+2} > y_{i+3}$.

Proof. A proof of this fact can be found in [1]. In [1], s_i is defined in terms of $x_{i-4}, \ldots, x_{i+5}$ and y_i is defined in terms of x_i and x_{i+1}. The formulation given here suits our application better since a vertex in a tree can only read the labels of its ancestors.

For the sake of completeness we include the definition of the functions y and f given in [1].

Let $y_i = y(x_{i+4}, x_{i+5})$ be the index of the rightmost bit where x_{i+4} and x_{i+5} differ and let b_i be the y_ith bit of x_{i+4}. Then, $0 \leqslant y_i \leqslant \lceil \log N \rceil - 1$. Define $s_i = f(x_i, \ldots, x_{i+9}) = Sel_i \vee Sel'_i$, where

$$
\begin{aligned}
A_i = true \quad & \text{iff} \quad y_i \leqslant \min(y_{i-1}, y_{i+1}),\\
Sel_i = true \quad & \text{iff} \quad A_i = true \text{ and either } A_{i-1} = A_{i+1} = false \text{ or } b_i = 1,\\
A'_i = true \quad & \text{iff} \quad y_i \geqslant \max(y_{i-1}, y_{i+1}) \text{ and } Sel_{i-1} = Sel_i = Sel_{i+1} = false,\\
Sel'_i = true \quad & \text{iff} \quad A'_i = true \text{ and either } A'_{i-1} = A'_{i+1} = false \text{ or } b_i = 1. \quad \square
\end{aligned}
$$

Proof of Theorem A (*continued*). We are now ready for the $O(\log^* n)$ algorithm. We assume that our almost-tree is given by an array *suc* and that each vertex has a unique name. We also assume that $suc[i] \neq i$ for all i. We shall basically construct a maximal dividing set by repeated application of Fact 2.2 above. More precisely, consider any path $v_0, v_1, \ldots$ starting in a leaf v_0. Each vertex v computes $s(v) = f(name(suc(v)) \ldots, name(suc^{(10)}(v)))$ and $t(v) = f(name(v), \ldots, name(suc^{(9)}(v)))$; vertices in the vicinity of the root behave differently. We add all vertices v with $s(v) = true$ to our maximal dividing set and delete them and their neighbors from further consideration, i.e., we delete a vertex w if either $s(suc(w))$ or $s(z)$ for some z with $w = suc(z)$. The former situation can be detected by concurrent read and the latter situation is easily detected since $t(w) = s(z)$ for every z with $w = suc(z)$. There is a catch, however. Although the t-value at a vertex w is identical to the s-value at all vertices z with $suc(z) = w$, the vertex w does *not* know whether it has any sons at all. This implies that we sometimes delete vertices from further consideration without adding a neighbor to the set of selected vertices. However, this only occurs close to leaves and, therefore, the length of a path of wrongly deleted vertices grows only by one in each iteration. Thus, after $\log^* n$ iterations, the selected vertices split the tree into subtrees of height at most $\log^* n$. In a second step we can then select further vertices in these subtrees. The details are as follows.

```
for k from 0 to log*n
do (*  we maintain the following invariant:
        (1) s(v) ⇒ del(v), del(suc(v)) and del(w) for all w with v = suc(w)
        (2) s defines an independent set, i.e., if s(v) and s(w), then v ≠ suc(w)
        (3) if ¬del(v), then name(v) ∈ [0..log^[k](n) − 1] and if ¬del(v), ¬del(suc(v)), then name(v)
            ≠ name(suc(v))
        (4) for all paths v_0, v_1, v_2, ... starting a leaf v_0 of T and all i: if del(v_i), then either s(v_i) or
            s(v_{i+1}) or there is a j ≤ k with del(v_{i-1}), del(v_{i-2}), ..., del(v_{i-j+1}) and either i − j < 0 or
            s(v_{i-j}); if ¬del(v_i) and k > 0, then ∃j ≤ log^[k](n): s(v_{i+j})   *)

    for all v with ¬del(v)
        pardo case
            d(v) > 10:  s(v) ← f(name(suc(v)), ..., name(suc^(10)(v)));
                        t(v) ← f(name(v), ..., name(suc^(9)(v)));
            d(v) = 10:  s(v) ← false;
                        t(v) ← f(name(v), ..., name(suc^(9)(v)));
            d(v) < 10:  s(v) ← "d(v) odd";
                        t(v) ← "d(v) even";
            esac;
        del(v) ← del(v) ∨ s(v) ∨ t(v) ∨ s(suc(v));
        if ¬del(v) then name(v) ← y(suc^(5)(v), suc^(6)(v)) fi
        odpar;
        k ← k + 1
od
```

Program 1.

We use variables $del(v)$, $sel(v)$ with $del(v) = sel(v) = false$ initially for all v. We furthermore use $d(v) = \min\{i;\, suc^{(i)}(v) = 0 \text{ or } del(suc^{(i)}(v))\}$ to denote the depth of vertex v in the almost-tree of undeleted vertices. We also define $\log^{[k]}(n)$ by $\log^{[0]}(n) = n$ and $\log^{[k+1]}(n) = \lceil \log(\log^{[k]}(n)) \rceil$ and $\log^*(n) = \min\{k;\, \log^{[k]}(n) \leqslant 1\}$. As mentioned above, we proceed in two steps.

***Step* 1** is given by Program 1.

We have to argue that the invariant is maintained. Let t_k, s_k, del_k, d_k denote the values of the variables t, s, del, d at the beginning of the kth iteration, $k \geqslant 0$. First note that for all vertices x with $\neg(del_k(x) \vee del_k(suc(x)))$ we have $t_{k+1}(suc(x)) = s_{k+1}(x)$ and hence $\neg(s_{k+1}(x) \wedge s_{k+1}(suc(x)))$. The last claim follows from part (a) of Fact 2.2 for $d_k(x) > 10$ and by construction for $d_k(x) \leqslant 10$. This shows that property (2) of Program 1 is maintained. Property (1) is maintained by the definition of del_{k+1}. Property (3) follows from part (c) of Fact 2.2. It remains to consider the fourth property. Consider any path $v_0, v_1, \ldots$ starting in a leaf v_0 of T and any vertex v_i on this path. Assume first that $del_{k+1}(v_i)$. If already $del_k(v_i)$, then property (4) obviously holds. So let us assume that $\neg del_k(v_i)$. Then, $s_{k+1}(v_i) \vee s_{k+1}(v_{i+1}) \vee t_{k+1}(v_i)$. If $s_{k+1}(v_i) \vee s_{k+1}(v_{i+1})$, then property (4) holds. It remains to consider the case that $\neg(s_{k+1}(v_i) \vee s_{k+1}(v_{i+1}))$ but $t_{k+1}(v_i)$. If $i = 0$, then property (4) trivially holds with $j = 1$. If $i > 0$, then we have to distinguish the cases $\neg del_k(v_{i-1})$ and $del_k(v_{i-1})$. In the former case, we have $t_{k+1}(v_i) = s_{k+1}(v_{i-1})$ and

```
for k from 0 to log*n
   do
      (*  we maintain the following invariant:
          (1) s defines an independent set, i.e., s(v) implies ¬s(suc(v))
          (2) for all v there is a j ≤ max(2, log*n − 2k) such that s(suc^(j)(v))   *)
      for all v
          pardo s(v) ← s(v) ∨ s(suc^(2)(v))
            if s(suc(v)) then s(v) ← false fi
          odpar
   od
for all v pardo if leaf(v) and ¬s(suc(v)) then s(v) ← true fi odpar
```

Program 2.

hence the property holds with $j=1$, and in the latter case the property immediately follows from the same property applied to $del_k(v_{i-1})$.

Assume next that $\neg del_{k+1}(v_i)$. We need to show that there is a $j \leqslant \log^{[k]}(n)$ such that $s_{k+1}(v_{i+j})$. If $d_k(v_i) \leqslant 9 + \log^{[k]}(n)$, then this is clearly the case since vertices of depth 9 are always selected. So let us assume that $d_k(v_i) \geqslant 10 + \log^{[k]}(n)$. Then, $t_{k+1}(v_i)$, $s_{k+1}(v_i), \ldots, s_{k+1}(v_{i+\log^{[k]}(n)-1})$ are all computed by the first clause of the case distinction. By part (b) of Fact 2.2 at least one of the values is *true*. Thus, we either established the invariant or derived a contradiction to $\neg del_{k+1}(v)$.

Upon termination of Step 1, all vertices are deleted (since, for $k = \log^* n$, each undeleted vertex must have name 0 and hence is deleted in this last iteration). Also, for each vertex v there is a $j \leqslant \log^* n$ such that $s(suc^{(j)}(v))$. This is *true* if $d_0(v) \leqslant \log^* n$ since the root is always selected and follows from the fourth invariant if $d_0(v) > \log^* n$.

***Step* 2** is given by Program 2.

Again we have to argue that the invariants hold. The first invariant is clearly maintained. For the second invariant, consider any vertex v and let j_k be the minimal j such that $s(suc^{(j)}(v))$ at the beginning of the kth iteration. If $j_k \geqslant 2$, then $j_{k+1} = j_k - 2$ and the invariant holds. If $j_k \leqslant 1$, then $j_{k+1} \leqslant 2$ and the invariant holds.

We conclude that upon termination of the loop there are no three consecutive vertices which were all not selected. Thus, the last statement computes a maximal dividing set. This completes the description of the O($\log^* n$) algorithm.

We turn next to an O($\log n$) algorithm which uses $n/\log n$ processors. We first execute the body of the loop of Program 1 once. This divides the almost-tree into trees of height at most $\log^{[1]}(n)$. Consider any path $v_i, v_{i+1}, \ldots, v_{i+l}$ of undeleted vertices. Then, either $0 \leqslant name(v_i) < \cdots < name(v_{i+l}) \leqslant \log^{[1]}(n)$ or $\log^{[1]}(n) \geqslant name(v_i) > \cdots > name(v_{i+l}) \geqslant 0$, i.e., the names along any such path form a monotone sequence. Moreover, if v and w are siblings, then $name(v) = name(w)$. We now step through the undeleted vertices in increasing order of names and select a vertex if its successor is not selected. This selects all vertices on paths along which the names are increasing and it selects every other vertex on paths along which the names are decreasing. We now step through the vertices once more, but this time in decreasing order of names, and 'deselect' a vertex if its successor is selected. This deletes every other vertex from the set of selected vertices on an increasing path and does not change the labelling on a decreasing path. Thus, in either case, we have computed a maximal dividing set. The details are given in Program 3.

It remains to prove the time and processor bound. The time bound is obvious. The processor bound directly follows from the corresponding bound in [1, Section 2.3]. □

```
for i from 0 to log^[1] n
    do for all v with name(v) = i
        pardo if ¬s(suc(v)) then s(v) ← true fi
        odpar
    od;
(* all nodes on an increasing path, i.e., name(v) < name(suc(v)) for all v, are selected and every other
   node on a decreasing path, i.e., name(v) > name(suc(v)) for all v, is selected *)

for i from log^[1] n downto 0
    do for all v with name(v) = i
        pardo if s(suc(v)) then s(v) ← ← false fi
        odpar
    od
(* along an increasing path the label of every other node is changed to "not selected", along a
   decreasing path the labelling is not changed *)
(* at this point we have computed a maximal dividing set except in the vicinity of the leaves *)
for all v
    pardo if leaf[v] and ¬s(suc(v)) then s(v) ← true fi
    odpar
```

Program 3.

Theorem B. *Maximal independent sets in undirected graphs with degree bound c can be computed on an* EREW-PRAM *in* $O(\log^* n)$ *time using* $O(n)$ *processors or in* $O(\log n)$ *time using* $n/\log n$ *processors.*

2.3. Remark. We assume that the graph is given by its adjacency list representation, i.e., for each vertex is a list of all neighbors. Furthermore, for each edge (v, w), the occurrence of w on v's adjacency list is linked with the occurrence of v on w's adjacency list.

Proof of Theorem B. We construct the maximal independent set in two phases. In the first phase we split the vertex set V into 2^c disjoint sets $V^{(1)}$, $V^{(2)}, \ldots,$ each set being an independent set in the original graph. In the second phase we shall then assemble a maximal independent set. We now describe the two phases in more detail.

***Phase* 1.** Phase 1 consists of c stages. We describe the first stage, the other stages being similar. Let $G = (V, E)$ be our graph. Consider the almost-tree $T = (V, A)$, where $A = \{(i, j);\ j$ is the first vertex on i's adjacency list$\}$, and let I be a maximal independent set in T. Let G' be the subgraph of G induced by I and let G'' be the subgraph induced by $V - I$. Then, the maximal degree of any vertex of G' or G'' is at most $c - 1$ since for $i \in I$ the edges in A incident to i connect i with vertices $j \notin I$ and for $i \notin I$ at least one of the edges in A incident to i connects i with a vertex $j \in I$. Also, every edge of G which is not an edge of either G' or G'' connects a vertex in I with a vertex in $V - I$ and hence an independent set of both G' or G'' is an independent set of G.

We apply the same procedure to graphs G' and G'' and obtain four graphs of maximal degree $c - 2$. Iterating this process c times, we obtain subgraphs $G^{(i)} = (V^{(i)}, E^{(i)})$, $1 \leqslant i \leqslant l = 2^c$, where

(1) $E^{(i)} = \emptyset$ and therefore $V^{(i)}$ is an independent set in $G^{(i)}$ and hence an independent set in G,

(2) $V^{(i)} \cap V^{(j)} = \emptyset$ for $i \neq j$ and $V = \bigcup_{i=1}^{l} V^{(i)}$.

```
I ← ∅
for i from 1 to l
do for all j ∈ V^(i)
    pardo if no neighbor of j is in I
            then add j to I
            fi
    odpar
od
```

Program 4.

***Phase* 2.** In Phase 2 we process the $V^{(i)}$'s sequentially and compute a maximal independent set (cf. Program 4).

Phase 2 clearly computes a maximal independent set (use induction to prove that I is a maximal independent set of the subgraph spanned by $V^{(1)} \cup \cdots \cup V^{(i)}$ after the ith iteration).

The running time of Phase 1 is determined by the running time of the independent set algorithm for almost-trees and the running time of Phase 2 is O(1). It remains to argue that all read and write conflicts can be avoided. This can be seen as follows. First note that in the algorithm only vertices which are adjacent in G communicate and hence a vertex has only c possible partners for communication. Hence, if each vertex numbers its neighbors from 1 to c (by going through its adjacency list and then using the link between an edge and its reverse), then the pth neighbor of any vertex i can use the time units t with $t \bmod c = p$ to read or write into i's storage. In this way no conflicts occur. □

3. Updating minimum spanning trees

In this section we treat the problem of updating minimum spanning trees upon the addition of a new vertex. Let $T = (V, A)$ be a minimum spanning tree (MST) of the weighted undirected graph $G = (V, V \times V, c)$; here, $c: V \times V \to \mathbb{R}$ is a weight function. Let $w \notin V$ be a vertex, let $V' = V \cup \{w\}$, and let $c': V' \times V' \to \mathbb{R}$ be an extension of c. The problem is to compute a minimum spanning tree of the graph $G' = (V', V' \times V', c')$. It is obvious that a tree $T' = (V', A')$ is a MST for G' iff T' is a MST for the graph $G'' = (V \cup \{w\}, A \cup \{(v, w); v \in V\})$, i.e., it suffices to consider the given MST $T = (V, A)$ together with the edges incident to the new vertex w. We assume that T is given by its adjacent lists.

Theorem C. *Minimum spanning trees can be updated on a* CRCW-PRAM *in* $O(\log n)$ *time using* $n/\log n$ *processors.*

Proof. We prove Theorem C by showing that updating a minimum spanning tree is strongly related to the problem of evaluating a certain expression involving the operators min and max. The expression is given by the spanning tree to be updated with every vertex replaced by a small formula of min's and max's. The result then follows from the expression evaluation technique of Cole and Vishkin [2].

In order to simplify the description of the algorithm we make the following assumptions:

(i) different edges have different weights,

(ii) the MST T is rooted.

The first property can be obtained by padding the weights (replace the weight $c(e)$ of an edge e by the pair $(c(e), \textit{name}(e))$, where $\textit{name}(e)$ is a unique name of edge e and then use the lexicographic ordering

on the pairs), the second property can be achieved by applying the Euler tour technique of Tarjan and Vishkin [8] together with the method of Cole and Vishkin [3] for list ranking. This takes exactly the time and processor bounds stated above.

For a vertex $v \in V$ let T_{vw} denote the subgraph induced by w and the subtree of T rooted at v. In order to update the given MST T we successively compute (starting with the leaves of T) for all $v \in V$,

$$T_v' = (V_v', E_v') \text{ a MST for } T_{vw}$$

and

$$e_v: \text{the maximum weight edge on the path in } T_v' \text{ from } w \text{ to } v.$$

For a leaf v we can compute T_v' and e_v very easily, namely:

$$e_v \leftarrow \{v, w\}, \qquad V_v' \leftarrow \{v, w\}, \qquad E_v' \leftarrow \{\{v, w\}\}.$$

For a nonleaf v, say with children $v_1, \ldots, v_k$, we compute

$$\text{(1)} \qquad e_i \leftarrow \begin{cases} e_{v_i} & \text{if } c(e_{v_i}) > c(\{v, v_i\}) \\ \{v, v_i\} & \text{otherwise} \end{cases} \quad \text{for } 1 \leqslant i \leqslant k$$

(* e_i is the maximum weight edge on the path from v to w through v_i *),

$$\text{(2)} \qquad c_i \leftarrow c(e_i), \quad 1 \leqslant i \leqslant k, \qquad c_0 \leftarrow c(\{v, w\}),$$

$$\text{(3)} \qquad e_v \leftarrow \begin{cases} \{v, w\} & \text{if } c_0 < \min\{c_i; 1 \leqslant i \leqslant k\}, \\ e_j & \text{if } c_j = \min\{c_i; 1 \leqslant i \leqslant k\} < c_0 \end{cases}$$

(* e_v is the maximum cost edge on the path from v to w in T_{vw} *),

$$\text{(4)} \qquad V_v' \leftarrow \{v\} \cup \bigcup_{1 \leqslant i \leqslant k} V_{v_i}', \qquad E_v' \leftarrow \bigcup_{1 \leqslant i \leqslant k} (E_{v_i}' \cup \{v, v_i\}) - \{e_i; 1 \leqslant i \leqslant k\} \cup \{e_v\}.$$

The correctness of this algorithm is easily proved by induction on the height of T, as follows.

Let us assume that T_{v_i}' and e_{v_i} $(1 \leqslant i \leqslant k)$ already have the desired properties, i.e.,

- T_{v_i}' is a MST for $T_{v_i w}$,
- e_{v_i} is the maximum weight edge on the path between v_i and w.

This implies that the graph induced by the edge set

$$\bigcup_{1 \leqslant i \leqslant k} (E_{v_i}' \cup \{v_i, v\}) \cup \{\{v, w\}\}$$

contains exactly $\binom{k+1}{2}$ simple cycles, all of them consisting of two paths from v to w. Such a path consists either of the single edge from v to w or runs through one of the v_i's. Thus, every edge in $\{e_i; 1 \leqslant i \leqslant k\} \cup \{\{v, w\}\}$ except the minimum cost edge in this set figures as a maximum weight edge on a simple cycle and hence must be excluded from the MST. This implies correctness.

We shall show next that the above algorithm can be viewed as evaluating a certain expression involving the operators min and max. First we assume that the indegree of every vertex v is at most two, i.e., $k \leqslant 2$ in the above algorithm, and later we shall show how to overcome this restriction. For a vertex v with sons v_i, $1 \leqslant i \leqslant k$, let f_v be the following k-ary function:

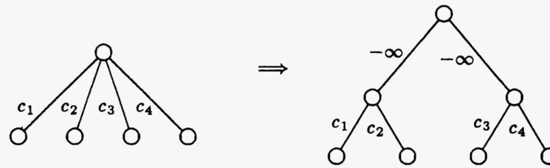

Fig. 2. Replacing a high-degree node by a binary tree.

$$k=0: \quad f_v = c(\{v, w\}),$$
$$k=1: \quad f_v = \lambda x.\min(c(\{v, w\}), \max(x, c(\{v, v_1\}))),$$
$$k=2: \quad f_v = \lambda x \lambda y.\min(c(\{v, w\}), \max(x, c(\{v, v_1\})), \max(y, c(\{v, v_2\}))).$$

The function f_v when applied to the cost of the edges e_i, $1 \leqslant i \leqslant k$, yields the cost of the edge e_v. Clearly, by using pairs of the form (cost of an edge, name of an edge) as arguments and values of these functions we could not only compute the cost of edge e_v but also the edge e_v itself. For simplicity, we concentrate on the cost. At this point we have associated with our current MST T an expression such that evaluating this expression and all its subexpressions computes the edge e_v and the edges e_i, $1 \leqslant i \leqslant k$ (cf. Step 1 above) for every vertex v. Once these edges are known, the MST is readily updated in O(1) time using $O(n)$ processors and hence in the time and processor bounds state above. Note that a vertex v deletes the edges $\{e_i; 1 \leqslant i \leqslant k\} - \{e_v\}$ from further consideration and that every edge is deleted only once. We therefore only have to label all edges as candidates initially and then let all vertices delete their set of edges in O(1) time. Also note that the adjacency list representation of the new spanning tree is readily computed in the stated time and processor bounds.

Before we turn to the problem of expression evaluation we need to discuss the case $k > 2$. We reduce it to the case $k \leqslant 2$ by building a balanced binary tree on top of the children of every vertex and giving the new edges cost $-\infty$ (cf. Fig. 2). Then, a MST of the modified graph directly yields a MST of the original graph. All of this can be done by using the techniques of Cole and Vishkin [1,2,3].

We finally turn to the problem of expression evaluation.

3.1. Definition. Let $\mathscr{F}_i$, $0 \leqslant i \leqslant 2$, be a set of i-ary functions. The set $\mathscr{F} = \mathscr{F}_0 \cup \mathscr{F}_1 \cup \mathscr{F}_2$ is called *nice* if, for all $a, b \in \mathscr{F}_0$, $f_1, f_2 \in \mathscr{F}_1$ and $F \in \mathscr{F}_2$: $f_1(a) \in \mathscr{F}_0$, $F(a, b) \in \mathscr{F}_0$, $f_2 \circ f_1 \in \mathscr{F}_1$, $\lambda x.F(a, x) \in \mathscr{F}_1$, and $\lambda x.F(x, a) \in \mathscr{F}_1$. Moreover, the compositions can be carried out in O(1) time.

3.2. Examples of nice classes of functions

(1) $\mathscr{F}_0 = \mathbb{R}$, $\mathscr{F}_2 = \{+, -, \times, /\}$, $\mathscr{F}_1 = \{\lambda x.(ax+b)/(cx+d);\ a, b, c, d \in \mathbb{R}\}$;

(2) $\mathscr{F}_0 = \mathbb{R}$, $\mathscr{F}_2 = \{\lambda x \lambda y.a + \max(x, y);\ a \in \mathbb{R}\}$, $\mathscr{F}_1 = \{\lambda x.a + \max(b, x);\ a, b \in \mathbb{R}\}$;

(3) $\mathscr{F}_0 = \mathbb{R}$, $\mathscr{F}_2 = \{\lambda x \lambda y.\min(a, \max(b, x), \max(c, x));\ a, b, c \in \mathbb{R}\}$,
$\mathscr{F}_1 = \{\lambda x.\min(a, \max(b, x));\ a, b \in \mathbb{R}\}$.

3.3. Remark. The function f_v defined above belongs to the third class.

The proof of Theorem C is now completed by applying the following result of Cole and Vishkin. □

Theorem D (Cole and Vishkin [2]). *Let $\mathscr{F}$ be a nice class of functions and let T be an expression tree of size n over $\mathscr{F}$. Then the expression and all its subexpressions can be evaluated on a* CRCW-PRAM *in* $O(\log n)$ *time using $n/\log n$ processors.*

Acknowledgment

We thank Torben Hagerup for many helpful discussions and the anonymous referee for his helpful comments.

References

[1] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. & Control* **70** (1) (1986) 32–53.

[2] R. Cole and U. Vishkin, Approximate and exact parallel scheduling with applications to list, tree and graph problems, *27th Conf. on Fundamentals of Computer Science* (1986) 478–491.

[3] R. Cole and U. Vishkin, Faster optimal parallel prefix sums and list ranking, *Inform. & Control*, to appear.

[4] A.V. Goldberg, S.A. Plotkin and G.E. Shannon, Parallel symmetry breaking in sparse graphs, *19th Symp. on Theory of Computing* (1987) 315–324.

[5] T. Hagerup, M. Chrobak and K. Diks, Parallel 5-coloring of planar graphs, *14th ICALP, Lecture Notes in Computer Science, Vol. 267* (Springer, Berlin, 1987) 204–312.

[6] M. Luby, A simple parallel algorithm for the maximal independent set problem, *17th Symp. on Theory of Computing* (1985) 1–10.

[7] S. Pawagi and I.V. Ramakrishnan, Parallel update of graph properties in logarithmic time, *Proc. 1985 Internat. Conf. on Parallel Processing* (1985) 186–193.

[8] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14** (4) (1985) 862–874.

[9] P. Varman and K. Doshi, A parallel vertex insertion algorithm for minimum spanning trees, *13th ICALP, Lecture Notes in Computer Science, Vol. 226* (Springer, Berlin, 1986) 424–433.