アルゴリズム 46－8
（1995．7．20）

A Parallel Algorithm for the Mincut Linear Arrangement of Binary Trees (Extended Abstract)

Sung Kwon KIM[1]

[1] Department of Computer Science
Kyungsung University
Pusan 608–736
Korea
ksk@csd.kyungsung.ac.kr.
fax: +82-51-628-6059

Supported in part by Korea Science and Engineering Foundation (No. 941-0900-002-1)

Abstract: We present a parallel algorithm for the mincut linear arrangement of binary trees. Given a binary tree with n vertices, our algorithm runs in $O(\log n)$ time using $O(n^{2.5}/\log n)$ processors in the CRCW PRAM.

A Parallel Algorithm for the Mincut Linear Arrangement of Binary Trees

Sung Kwon KIM[1]

[1] Department of Computer Science
Kyungsung University
Pusan 608–736
Korea

本文では，2分木の最小カット線形配置を求める並列アルゴリズムを与える．n 点の2分木が与えられたときに，アルゴリズムは CRCW PRAM 上で $O(n^{2.5}/\log n)$ 個のプロセッサを用いて $O(\log n)$ 時間で終了する．

1 Introduction

Given a graph $G = (V, E)$ with $|V| = n$, a *linear arrangement* of G is a one-to-one mapping π from V to $\{1, 2, \ldots, n\}$. A linear arrangement π embedds G on the real line so that the vertices locate at the first n positive integers and the edges connect corresponding vertices. There are several well-studied linear arrangement problems; for a survey of their complexity results, see, e.g., [D].

We consider one of the linear arrangement problems, called the MINCUT problem. The MINCUT problem is to find a linear arrangement π that minimizes $\gamma_\pi(G) = \max_{1 \le i \le n}(\mid \{(u, v) \in E : \pi(u) \le i < \pi(v)\} \mid)$, which is called the *cut* of G under π. Denote the minimum cut (or mincut) of G by $\gamma(G)$. The MINCUT problem for general graphs is NP-complete [G], while the problem is solvable in $O(n \log n)$ time for trees of unbounded degree [Y] and in $O(n \log^{d-2} n)$ time for degree d trees [C]. For degree three trees, both algorithms take $O(n \log n)$ time, but the latter is simpler. The MINCUT problem for trees of unbounded degree can be solved in $O(\log^2 n)$ time using $O(n^2)$ processors in the CREW PRAM [D1]; this algorithm makes use of ideas of the sequential algorithm of [Y] and applies the parallel tree contraction [J] in a novel way. The algorithm still takes the same complexity bounds for degree three trees.

In this paper, we present a parallel algorithms for the MINCUT problem on trees of degree three, which requires $O(\log n)$ time using $O(n^{2.5}/\log n)$ processor in the CRCW PRAM. Our algorithm is a non-straightforward parallel implementation of the $O(n \log n)$ time sequential algorithm of [C] and works in two stages. In the first stage (described in Section 3), it computes information, including $\gamma(T)$ among others, to be used in the second stage, by using the parallel tree contraction technique [J]. This stage needs $O(n^{2.5}/\log n)$ processors. The second stage (described in Section 4) consisting of $O(\log n)$ phases actually assigns $1, 2, \ldots, n$ to the vertices of T so that $\gamma(T)$ is achieved. Careful implementation is required to make each phase of the stage take $O(1)$ time using $O(n^2)$ processors.

2 Review

This section reviews the result of Chung, Makedon, Sudborough and Turner [C] concerning the mincut linear arrangement of degree three trees.

Let $T = (V, E)$ be a tree and let $\{u, v_1, \ldots, v_s\} \subseteq V$. Define $T(u, v_1, \ldots, v_s)$ to be the maximal subtree of T that contains u but none of $v_1, \ldots, v_s$. The following theorem is crucial for our algorithm [C].

Theorem 1. *$\gamma(T) \le k$ if and only if every vertex u of degree ≥ 2 has neighbors v_1, v_2 such that $\gamma(T(u, v_1, v_2)) \le k - 1$.*

Proof: We give a slightly modified version of the proof of [C] for the *if* part only as this gives an idea for our algorithm.

Consider two cases.

Case (i): Suppose that every vertex u of degree ≥ 2 has a neighbor x such that $\gamma(T(u, x)) \le k - 1$. Note that this implies the *if* condition that every vertex u of degree ≥ 2 has neighbors, v_1, v_2 such that $\gamma(T(u, v_1, v_2)) \le k - 1$. Choose a vertex y_1 of degree ≥ 2. Then, y_1 has a neighbor y_2 such that $\gamma(T(y_1, y_2)) \le k - 1$. If y_2 is not a leaf, then it has a neighbor $y_3 \ne y_1$ such that $\gamma(T(y_2, y_1, y_3)) \le k - 1$. If $\gamma(T(y_2, y_1, z)) \ge k$ for all neighbors $z \ne y_1$ of y_2, then there would be no neighbor x of y_2 such that $\gamma(T(y_2, x)) \le k - 1$. Similarly, if y_3 is not a leaf, then it has a neighbor $y_4 \ne y_2$ such that $\gamma(T(y_3, y_2, y_4)) \le k - 1$. Let $P = (y_1, \ldots, y_r)$ be a path constructed in this way, with a leaf y_r. $T \sim P$ [1] consists of trees $T(y_1, y_2)$, $T(y_{i+1}, y_i, y_{i+2})$ for $1 \le i \le r - 2$, and the leaf $T(y_r, y_{r-1}) = y_r$, each of which has $\gamma(\cdot) \le k - 1$. So, $\gamma(T) \le k$.

Case (ii): Suppose that there is some vertex c such that $\gamma(T(c, x)) \ge k$ for all neighbors x of c. By the *if* condition, c has neighbors y_1, z_1 such that $\gamma(T(c, y_1, z_1) \le k - 1$. Let $c = y_0 = z_0$. If y_1 is not a leaf, then it has a neighbor $y_2 \ne y_0$ such that $\gamma(T(y_1, y_0, y_2)) \le k - 1$. To prove this, we need to show that $y_0 \in \{v_1, v_2\}$, where v_1 and v_2 are neighbors of y_1 such that $\gamma(T(y_1, v_1, v_2)) \le k - 1$. If $y_0 \notin \{v_1, v_2\}$, then there is a contradiction between

[1] $A \sim B$ removes from A the edges in B but not their end vertices; $A - B$ removes both the vertices in B and the edges incident on them.

$\gamma(T(y_1, v_1, v_2)) \leq k-1$ and $T(y_0, y_1) \geq k$ because $T(y_1, v_1, v_2) \supset T(y_0, y_1)$. Similarly, if y_2 is not a leaf, then it has a neighbor y_3 such that $\gamma(T(y_2, y_1, y_3)) \leq k-1$. Continuing in this way we can construct a path $P_1 = (y_0, y_1, \ldots, y_r)$ such that $\gamma(T(y_{i+1}, y_i, y_{i+2})) \leq k-1$ for $1 \leq i \leq r-1$, and y_r is a leaf.

A similar path $P_2 = (z_0, z_1, \ldots, z_s)$ can be constructed. Let $P = P_1 \cup P_2$. $T \sim P$ consists of trees $T(c, y_1, z_1)$, $T(y_{i+1}, y_i, y_{i+2})$ for $0 \leq i \leq r-2$, and $T(z_{i+1}, z_i, z_{i+2})$ for $0 \leq i \leq s-2$, each of which has $\gamma(\cdot) \leq k-1$. So, $\gamma(T) \leq k$. □

Lemma 1. [C] *For a tree T of degree d with n vertices, $\gamma(T) < (d/2)\log n + 1$.*

So far, we have assumed that trees are of unbounded degree. In the remaining part of this paper, we will assume that trees T are of degree three. Choose a vertex r of degree one or two and let T be rooted at r. T is now a binary tree. $T[u]$ for a vertex u of T denotes the subtree of T rooted at u. So, $T = T[r]$. For an edge $e = (u, v)$ (v is the parent of u), define $T[e] = T[u] \cup \{e\}$, i.e., $T[e]$ is $T[u]$ plus vertex v and edge e. e is a *child edge* of v and the *parent edge* of u; and u and v are the *child vertex* and *parent vertex* of e, respectively. For descendants $v_1, \ldots, v_s$ of vertex u, $T[u, v_1, \ldots, v_s] = T[u] - (\cup_{i=1}^{s} T[v_i])$. For descendants $v_1, \ldots, v_s$ of edge e, $T[e, v_1, \ldots, v_s] = T[e] - (\cup_{i=1}^{s} T[v_i])$.

A path P of a tree T is called a *basic path* if $T \sim P$ consists of trees T_i with $\gamma(T_i) \leq \gamma(T) - 1$ for all i.

A vertex c in T is *k-critical* if c has the child edges e_1, e_2 and $\gamma(T[e_i]) = k$ for $i = 1, 2$. c is k-critical in the sense that if $\gamma(T[w]) = k$ for an ancestor w of c, then any basic path of $T[w]$ must contain e_1, e_2.

The following theorem is a restatement of Theorem 1 by using the k-criticalness. (Proof omitted.)

Theorem 2. *Let T be a binary tree with root r. Let k be a positive integer. (i) If T has no ℓ-critical vertex for $\ell \geq k$, then $\gamma(T) \leq k$. (ii) If T has no ℓ-critical vertex for $\ell > k$ and exactly one k-critical vertex c such that $\gamma(T[r, v_1, v_2]) \leq k-1$ for two children v_1, v_2 of c, then $\gamma(T) = k$. (iii) If either T has more than one k-critical vertex or T has exactly one but the one does not satisfy the condition stated in (ii), then $\gamma(T) > k$.*

Let T be a binary tree with root r. To compute $\gamma(T)$, it is important to locate ℓ-critical vertices of T for $\ell \geq 1$. Define $\Gamma(T)$ as in [C] to be a decreasing sequence of integers $[a_1, \ldots, a_s]$ with $a_1 > \cdots > a_s \geq 0$ if T has vertices $u_1, \ldots, u_{s-1}$ and u_i has children y_i, z_i such that

1. $\gamma(T[r, y_1, z_1, \ldots, y_{i-1}, z_{i-1}]) = a_i$ for $1 \leq i \leq s$,
2. u_i is a_i-critical in $T[r, y_1, z_1, \ldots, y_{i-1}, z_{i-1}]$ for $1 \leq i \leq s-1$,
3. $T[r, y_1, z_1, \ldots, y_{s-1}, z_{s-1}]$ has no a_s-critical vertex, and
4. $[a_1, \ldots, a_s]$ is lexicographically minimum among those satisfying the three conditions above.

For an empty tree T, $\Gamma(T) = [\,]$.

To explain intuitively what $\Gamma(T)$ is, we give the following "wrong" algorithmic definition of $\Gamma(T)$: Initially, $i = 1$. Compute $\gamma(T)$, and set $a_i = \gamma(T)$. Determine whether T contains an a_i-critical vertex v. If yes, set $u_i := v$; $T := T[r, y_i, z_i]$; $i := i + 1$; and repeat the procedure. If no, stop.

For example, if T is a single-edge tree, then $\Gamma(T) = [1]$ as $\gamma(T) = 1$ and it has no 1-critical vertex. For a single-vertex tree T, $\gamma(T) = 0$ and $\Gamma(T) = [0]$. For a three-vertex complete binary tree, $\Gamma(T) = [1, 0]$ as the root is 1-critical.

We compute $\Gamma(T[u])$ in a bottom-up fashion by calling function Gamma of [C] in Figure 1 for all internal vertices u of T, after assigning [0] to all leaves of T. For an internal vertex u with children v_1, v_2, and child edges $e_1 = (v_1, u), e_2 = (v_2, u)$, given $\Gamma(T[v_i])$ for $i = 1, 2$, the first part (lines (3)–(13)) of function Gamma computes $\Gamma(T[e_i])$ from $\Gamma(T[v_i])$, and its second part (lines (14)–(23)) combines $\Gamma(T[e_1])$ and $\Gamma(T[e_2])$ to obtain $\Gamma(T[u])$. For correctness of function Gamma, see [C].

Each call of Gamma takes $O(\log n)$ time as S_i contains at most $\gamma(T) = O(\log n)$ elements. So, time to compute $\Gamma(T[u])$ for all vertices u of T is $O(n \log n)$.

To actually arrange the vertices of T, we make use of $\Gamma(T[u])$ computed above. Note that $\gamma(T) = \max \Gamma(T)$. Assume that $\gamma(T) = k$. Find a basic path $P = (v_1, \ldots, v_m)$ of T (by

```
(1)  function Gamma(S_1, S_2)  {S_i = Γ(T[v_i])}
(2)    { Assume that S_1 ≥ S_2.}
(3)    if min S_1 ≠ 0 then H_1 := S_1
(4)    else begin
(5)      j := min{i > 0 | i ∉ S_1}
(6)      H_1 := [j] ∪ [i > j | i ∈ S_1]
(7)      end
(8)    if S_2 = [ ] then return H_1
(9)    if min S_2 ≠ 0 then H_2 := S_2
(10)   else begin
(11)     j := min{i > 0 | i ∉ S_2}
(12)     H_2 := [j] ∪ [i > j | i ∈ S_2]
(13)     end
(14)   {H_i = Γ(T[e_i]), i = 1, 2.}
(15)   if H_1 ∩ H_2 = ∅ then begin
(16)     h := max{min H_1, min H_2}
(17)     return [i ≥ h | i ∈ H_1 ∪ H_2]
(18)     end
(19)   h := max H_1 ∩ H_2
(20)   if h = min H_1 = min H_2 then
(21)     return [0] ∪ [i ≥ h | i ∈ H_1 ∪ H_2]
(22)   i := min[j > h | j ∉ H_1 ∪ H_2]
(23)   return [i] ∪ [j > i | j ∈ H_1 ∪ H_2]
(24)   end
```

Figure 1: Function for computing $\Gamma(T[u])$ from $\Gamma(T[v_1])$ and $\Gamma(T[v_2])$ [C].

using the idea in the proof of Theorem 1) such that $T \sim P$ consists of subtrees T_i ($v_i \in T_i$) satisfying $\gamma(T_i) \leq k-1$. For $1 \leq i \leq m$, assign $\{n_{i-1}+1, \ldots, n_i\}$ to T_i in a recursive way, where $n_0 = 0$ and $n_i = |T_1| + \cdots + |T_i|$. Time for the arrangement is $O(n)$.

We finish our review of the algorithm of [C].

We will implement the algorithm in the CRCW PRAM. Our parallel implementation also consists of two stages: namely, the first stage (section 3) of computing $\Gamma(T[u])$ for all vertices u of T, and the second stage (section 4) of arranging the vertices of T.

The implementation is non-trivial but requires a non-straightforward application of the parallel tree contraction and a careful bookkeeping of several labels in recursive calls of the algorithm.

```
(1)  function Delta(δ_1, δ_2)
(2)    { Assume that δ_1 ≥ δ_2. }
(3)    if δ_1 is even then θ_1 := δ_1
(4)    else θ_1 := δ_1 + 1
(5)    if δ_2 = 0 then return θ_1
(6)    if δ_2 is even then θ_2 := δ_2
(7)    else θ_2 := δ_2 + 1
(8)    {θ_i represents Γ(T[e_i]), i = 1, 2.}
(9)    if (θ_1 and θ_2) = 0 then begin
(10)     h := max(rmo(θ_1), rmo(θ_2))
(11)     return (θ_1 or θ_2) and mask(h)
(12)     end
(13)   h := lmo(θ_1 and θ_2)
(14)   if h = rmo(θ_1) = rmo(θ_2) then begin
(15)     return 1 + (θ_1 or θ_2)
(16)     end
(17)   i := rmo((not(θ_1 or θ_2)) and
         mask(h + 1))
(18)   return 2^i + ((θ_1 or θ_2) and
         mask(i + 1))
(19) end
```

Figure 2: Function for computing $\delta(u)$ from $\delta(v_1)$ and $\delta(v_2)$.

3 Computing the cut and other information

By the definition of $\Gamma(\cdot)$, $\gamma(T) = a_1$. By Lemma 1, $\gamma(T) < 1.5 \log n + 1$ for trees T of degree three, and thus $0 \leq a_i \leq \lfloor 1.5 \log n \rfloor + 1$ for $a_i \in \Gamma(T)$. Let $M = \lfloor 1.5 \log n \rfloor + 1$. Then, $\Gamma(T)$ can be represented as a bit-vector $b_M b_{M-1} \cdots b_1 b_0$ such that $b_i = 1$ if and only if $i \in \Gamma(T)$.

We will assign to each vertex u of T an unsigned integer $\delta(u)$ defined as $\delta(u) = \sum_{i=0}^{M} b_i \cdot 2^i$, which represents $\Gamma(T[u])$. Note that $0 \leq \delta(u) \leq 2^{M+1} - 1 \leq 4n^{1.5}$.

Let u be a vertex with two children v_1, v_2. Then $\delta(u) = \text{Delta}(\delta(v_1), \delta(v_2))$, where function Delta is described in Figure 2. For leaves u, $\delta(u) = 1$ as $\Gamma(T[u]) = [0]$.

Function Delta uses bit-vectors to represent Γ's while function Gamma uses sorted lists of integers. In Delta, $\text{rmo}(i)$ (resp., $\text{lmo}(i)$) returns the bit position of the rightmost (resp., leftmost) "1" in the $M+1$-bit binary representation of an unsigned integer i, and $\text{mask}(h)$

returns the $M+1$-bit string such that its rightmost h bits are all zero and the remaining bits are all one, i.e., $\mathrm{mask}(h) = 2^{M+1} - 1 - (2^h - 1)$. The operators, **and**, **or**, **not**, are all bitwise.

Correctness of function Delta is immediate from that of function Gamma. (Details omitted.)

Computing $\mathrm{rmo}(i)$ for $0 \leq i \leq 2^{M+1} - 1$ (i requires $M+1$ bits) can be done as follows: Precompute $\mathrm{rmo}(j)$ for all $0 \leq j < n$. Divide the binary representation of i ($M+1$ bits long) into the lower part, i_1, consisting of the rightmost $\lceil \log n \rceil$ bits and the upper part, i_2, consisting of the remaining $M + 1 - \lceil \log n \rceil$ bits. If $i_1 = 0$ then $\mathrm{rmo}(i) = \mathrm{rmo}(i_2) + \lceil \log n \rceil$. Otherwise $\mathrm{rmo}(i) = \mathrm{rmo}(i_1)$.

Precomputing $\mathrm{rmo}(j)$ for all $0 \leq j < n$ can be done in $O(\log n)$ time using $O(n/\log n)$ processors. Then $\mathrm{rmo}(i)$ for $0 \leq i \leq 2^{M+1} - 1$ can be computed in constant time using a single processor. In an analogous way, $\mathrm{lmo}(i)$ can be computed in constant time using a single processor if a precomputed table $\mathrm{lmo}(j)$ for $0 \leq j < n$ is available.

Combining all together, we can say that one call of $\mathrm{Delta}(\delta_1, \delta_2)$ takes constant time using a single processor if precomputation has been done to prepare $\mathrm{lmo}(j)$ and $\mathrm{rmo}(j)$ for $0 \leq j < n$.

The *parallel tree contraction* [J] will be used to compute $\delta(u)$ for all vertices u of T. The parallel tree contraction contracts a binary tree T to a single vertex by processing a logarithmic number of intermediate binary trees $T_i = (V_i, E_i)$, $i = 0, 1, \ldots, t$ with $t = O(\log n)$. Initially, $T_0 = T$, and contract T_i to obtain T_{i+1}. Finally, T_t consists of a single vertex.

Let z be a leaf of T_i with parent u, grandparent x, and sibling y. The primitive operation, called *rake*, applied to z removes both u and z, and links y as a child of x. By carefully applying rake operations to a subset of leaves of T_i, one can contract T_i to T_{i+1} and decrease the number of leaves from ℓ to $\ell/2$, where ℓ is the number of leaves of T_i.

We can show that a rake operation applied to a leaf of T_i can be done in constant time using $O(n^{1.5})$ processors. (Details omitted.)

Theorem 3. *Given a binary tree T, computing $\delta(u)$ for all vertices u of T can be done in $O(\log n)$ time using $O(n^{2.5}/\log n)$ processors in the EREW PRAM.*

4 Linear arrangement

In the previous section, given a binary tree T, we have assigned to each vertex u of T an unsigned integer $\delta(u)$ which contains information needed to arrange the vertices of T in an optimal way. The basic idea of arranging T is in the proof of Theorem 1. So, our linear arrangement algorithm is basically the same as that of [C], but requires careful implementation details, including representations of trees and updating several labels of edges and vertices.

We first compute $\delta(e)$ for all edges e of T by doing the following:

if $\delta(u)$ is even **then** $\delta(e) := \delta(u)$

else $\delta(e) := \delta(u) + 1$

where e is the parent edge of u.

We will assume that either T consists of a single vertex only or the root of T has only one child. This assumption automatically holds in all subtrees of T appearing during the execution of our arrangement algorithm as will be clear later. If e is the child edge of the root of T, then $\delta(e)$ represents $\Gamma(T)$. In the remaining of this section, we will use the $\delta(\cdot)$ for the edges of T only but no longer those for the vertices of T.

Remember that $\gamma(T[e])$ is the leftmost bit position of $\delta(e)$ containing a "1".

4.1 Overview of the algorithm

We describe an overview of our algorithm for arrangement.

Step 1: (Preprocessing) Let T be a binary tree with $\delta(e)$ for all edges e of T whose root r has less than two children. We assume that the vertices of T are stored in an array $A[1..n]$ such that $A[i]$ contains the vertex with preorder number i. Each vertex will be responsible for its parent edge, i.e., a vertex has all information of its parent edge. If T consists of a single vertex r, then assign 1 to r and stop. Otherwise, do some preprocessing: Compute $\mathrm{ND}(u)$, the number of descendants of u including u itself, for all vertices u of T; and mark edges of T and compute $\mathrm{HEAD}(e)$ for all edge e of T as explained in section 4.2.

Find a basic path of T and denote it by $P = (v_1, \ldots, v_m)$. Set START:= v_1.

Step 2: Let $T \sim P$ consist of subtrees $T_1, \ldots, T_m$ where T_i contains v_i. Find a basic path of T_i and denote it by $P_i = (w_{i,1}, \ldots, w_{i,\ell_i})$. If T_i consists of a single vertex v_i, then $\ell_i = 1$ and $w_{i,1} = v_i$. Set SUCC$(w_{i,j}) := w_{i,j+1}$ for $1 \leq j < \ell_i$. Set SUCC$(w_{i,\ell_i}) := w_{i+1,1}$, if $1 \leq i \leq m-1$.

Step 3: Recursively repeat Step 2 for $T_i \sim P_i$ if $|T_i| \geq 2$.

Step 4: (Postprocessing) All vertices in T are linked as a linear linked list SUCC$(\cdot)$ with starting vertex START. Apply the list ranking [J] to number the vertices.

Theorem 4. *The algorithm above can be performed in $O(\log n)$ time using $O(n^2)$ processors in the CRCW PRAM.*

Proof: Step 1 (see section 4.2) and step 4 can be easily executed in $O(\log n)$ time using $O(n)$ processors. Since $\gamma(T) = O(\log n)$ for a binary tree T by Lemma 1, the algorithm recurses $\gamma(t)$ times at step 3. By Lemma 2 below each execution of Step 2 takes constant time using $O(n^2)$ processors. Hence, the whole algorithm runs in $O(\log n)$ time using $O(n^2)$ processors in the CRCW PRAM. □

Lemma 2. *Execution of Step 2 for each recursion can be done in constant time using $O(n^2)$ processors in the CRCW PRAM.*

Lemma 2 will be proved in the remainder of this section.

4.2 Finding basic paths

As our algorithm proceeds, T will be partitioned into several subtrees of different sizes; the algorithm stops when all subtrees consist of a single vertex. We will first explain how to find a basic path of a subtree of T. Let T' with $\gamma(T') = k$ be a subtree, rooted at r', appeared during the execution of the algorithm. We have two cases to consider by Theorem 2.

Case (i): T' has no k-critical vertex.

Case (ii): T' has only one k-critical vertex c and $\gamma(T'[r', v_1, v_2]) \leq k-1$ where c has children v_1 and v_2.

Cases (i) and (ii) will be used in the remaining part of this section to refer the cases stated above.

In either of cases (i) and (ii) we need to find a basic path P of T' such that $T' \sim P$ consists of subtrees T'' satisfying $\gamma(T'') \leq k-1$ as explained in Section 2.

Each vertex with two child edges e_1, e_2 compares $\gamma(T'[e_1])$ and $\gamma(T'[e_2])$ and marks the edge with a larger value (tie breaks arbitrarily). Each single-child vertex marks its child edge. Then T' contains several marked paths (each marked path has as its end vertices an internal vertex and a leaf). Each marked edge e has HEAD(e) which points to the vertex closest to the root in the marked path which e belongs to.

Case (i) Let P be the marked path containing root r'. Then, P is a root-to-leaf path. We will show that each subtree T'' in $T' \sim P$ has $\gamma(T'') \leq k-1$. Let u be a vertex in P. If u has only one child in T', then the subtree T'' containing u consists of u itself; thus $\gamma(T'') = 0$. If u has two children, then let e_1 and e_2 be its marked and unmarked child edges, respectively. Then, $T'[e_2]$ is a subtree in $T' \sim P$. Since $\gamma(T'[e_1]) \geq \gamma(T'[e_2])$ and T' has no k-critical vertex, $\gamma(T'[e_2]) < k$. So, P is a basic path of T'.

Each marked edge e can determine whether it belongs to P or not by comparing its HEAD(e) with r'.

Case (ii) In this case T' has only one k-critical vertex c and $\gamma(T'[r', v_1, v_2]) \leq k-1$ where v_1, v_2 are the children of c. Since c is k-critical, $\gamma(T'[e_1]) = \gamma(T'[e_2]) = k$ for its two child edges e_1, e_2. Let $e_1 = (v_1, c)$ (resp., $e_2 = (v_2, c)$) be the marked (resp., unmarked) child edge of c. Unmark the parent edge of c (it is easy to see that this edge was marked). Mark e_2. Both of c's child edges are now marked and its parent edge is now unmarked. Let P be the marked path containing c. P is a leaf-to-leaf path. The subtree in $T' \sim P$ containing c is $T'[r', v_1, v_2]$, whose $\gamma(\cdot)$ is $\leq k-1$. For the other subtrees T'' in $T' \sim P$, it can be shown in a similar way as in case (i) that $\gamma(T'') \leq k-1$. Thus P is a basic path.

We will show how a marked edge e can determine whether it belongs to P or not. Since P is a leaf-to-leaf path containing c, it can be partitioned at c into two descending subpaths P_1, P_2. Assume that $e_1 \in P_1$ and $e_2 \in P_2$. If HEAD$(e) = v_2$, then $e \in P_2$. If HEAD$(e) =$ HEAD(e_1) and e is further from the root than

e_1 is, then $e \in P_1$. Since the preorder numbers of the vertices of T' are known (see section 4.3), it is easy to determine which of e and e_1 is further from the root.

4.3 Updating tree representations

As mentioned earlier in step 1 of our algorithm overview, the vertices of T are stored in an array $A[1..n]$ such that $A[i]$ contains the vertex with preorder number i. Each subtree T' appeared during the execution of our algorithm will be assigned a triple of integers (α, β, ω) such that

1. $A[\alpha]$ contains the root of T';
2. $A[\beta..\omega]$ contains the other vertices of T' in preorder ($A[\beta]$ contains the vertex with preorder number 2 in T', $A[\beta+1]$ one with 3, and so on); and
3. $\beta = \omega = 0$, if T' consists of a single vertex.

For T, we have $\alpha = 1$, $\beta = 2$ and $\omega = n$.

Let $\gamma(T') = k$ and its triple be (α, β, ω) satisfying the conditions above. Let P be a basic path of T' computed in section 4.2. We will show how each subtree T'' in $T' \sim P$ can be assigned a triple of integers in constant time using $O(|T'|)$ processors in the CREW PRAM.

Case (i) T' has no k-critical vertex and P is a root-to-leaf path. Each subtree T'' in $T' \sim P$ either consists of a single vertex or is a tree with two or more vertices whose root has only one child. If T'' consists of a single vertex v, then T'' has $(\alpha_1, 0, 0)$ such that $A[\alpha_1] = v$. If T'' consists of root v, its child w and the descendants of w, then T'' has $(\alpha_1, \beta_1, \omega_1)$ such that $A[\alpha_1] = v$, $A[\beta_1] = w$ and $\omega_1 = \beta_1 + \mathrm{ND}'(w) - 1$, where $\mathrm{ND}'(w)$ is the number of the descendants of w in T'. By a property of preorder traversal, the descendants of w in T' appear consecutively in $A[\beta_1..\omega_1]$.

Case (ii) T' has the unique k-critical vertex c with two children v_1, v_2, and P is a leaf-to-leaf path. Let $T_i = T'[v_i]$ for $i = 1, 2$. Let $T_3 = T'[r', v_1, v_2]$, where r' is the root of T'. T_3 is one of the subtrees in $T' \sim P$.

We first show that T_3 can be assigned a triple of integers. Suppose that $A[\rho - 1] = c$ and the descendants of c (i.e., the vertices of $T_1 \cup T_2$) are stored consecutively in $A[\rho..\sigma]$ for integers $\beta \le \rho < \sigma \le \omega$. This is true because the vertices of T' are stored in $A[\alpha]$ and $A[\beta..\omega]$ in the order of their preorder traversal and the descendants of c appear consecutively. If $\sigma = \omega$ (i.e., if no vertex of T_3 has preorder number greater than that of c), then T_3 has a triple $(\alpha, \beta, \rho - 1)$ and we are done. So, assume that $\sigma < \omega$. Some vertices in T_3 have preorder number greater than that of c. The vertices in $A[\beta..\rho-1]$ and $A[\sigma+1..\omega]$ belong to T_3, while those in $A[\rho..\sigma]$ belong to $T_1 \cup T_2$. Move $A[\rho..\sigma]$ to $A[\rho - \sigma + \omega..\omega]$ and move $A[\sigma+1..\omega]$ to $A[\rho..\rho-\sigma+\omega-1]$. Then the root of T_3, r', is in $A[\alpha]$ and the other vertices of T_3 are in $A[\beta..\rho-\sigma+\omega-1]$. It is not difficult to see that those in $A[\beta..\rho-\sigma+\omega-1]$ appear consecutively in the order of preorder traversal of T_3. Thus, T_3 has a triple $(\alpha, \beta, \rho-\sigma+\omega-1)$. Data movement in $A[\rho..\omega]$ can be done in constant time using $O(|T'|)$ processors.

Now, T_1 has a new triple $(\rho - \sigma + \omega, \rho - \sigma + \omega + 1, \mu)$, and T_2 has a new triple $(\mu + 1, \mu + 2, \omega)$, where $\mu = \rho - \sigma + \omega + \mathrm{ND}'(v_1) - 1$. For $i = 1, 2$, since $P \cap T_i$ is a root-to-leaf path in T_i, it is easy to see that each subtree T'' of $T_i \sim P$ can be assigned a triple of integers in a similar way as in case (i).

We have showed that all subtrees appeared during the execution of the algorithm can be assigned by triples of integers. All vertices of a subtree except its root are stored in the order of their preorder sequence in a subarray of A. This is important for each recursive execution of our algorithm to run in constant time.

4.4 Updating δ's

Initially, we have $\delta(e)$ (representing $\Gamma(T[e])$) for all edges e of T. As before, let T' be a subtree appeared during the execution of the algorithm. Assume that $\delta'(e)$ (representing $\Gamma(T'[e])$) for all edges e of T' are available. We will show that, for each subtree T'' in $T' \sim P$, $\delta''(e)$ (representing $\Gamma(T''[e])$) for all edges e in T'' can be obtained in constant time using $O(|T'|)$ processors. Let $\gamma(T') = k$.

Case (i) $\delta''(e) = \delta'(e)$ for all edges e in $T' \sim P$.

Case (ii) Let Q be the path between c and the root of T'. For the edges e in $T' \sim P$,

$$\delta''(e) = \begin{cases} \delta'(e) - 2^k & \text{if } e \in Q, \\ \delta'(e) & \text{otherwise.} \end{cases}$$

4.5 Updating HEAD's

We now show how to update HEAD's. We assume all definitions in the previous subsections.

Case (i) For the marked edges e of $T' \sim P$, $\mathrm{HEAD}''(e) =$

$$\begin{cases} \text{parent of } \mathrm{HEAD}'(e) & \text{if the parent of } \mathrm{HEAD}'(e) \text{ is in } P, \\ \mathrm{HEAD}'(e) & \text{otherwise.} \end{cases}$$

Case (ii) Let $T_i = T'[v_i]$ for $i = 1, 2$. Let $T_3 = T'[r', v_1, v_2]$, where r' is the root of T'. For those edges e in $T_1 \cup T_2 \sim P$, $\mathrm{HEAD}''(e)$ can be assigned in a similar way as in case (i).

For those edges e in T_3, we need to be careful to determine $\mathrm{HEAD}''(e)$. Remember from case (ii) of section 4.4 that $\delta''(e) = \delta'(e) - 2^k$ for the edges $e \in Q$. For all vertices $u \in Q$, we should do marking again: compare the $\gamma(\cdot)$ values of their children and mark the edge with a larger value. We do not need re-marking for the edges $e \in T_3 - Q$ because $\delta''(e) = \delta'(e)$.

To update $\mathrm{HEAD}''(e)$ for $e \in T_3$, we first consider the edges in Q. Note that all edges in Q were marked in T'. After the re-marking above, some edges in Q are still marked, and the others are unmarked (instead, their sibling edges are now marked; we call these sibling edges *newly marked* edges).

As mentioned in section 4.3, T_3 is assigned a triple of integers (α, β, ω). The root is in $A[\alpha]$ and the other vertices are in $A[\beta\,..\,\omega]$ in the order of preorder traversal. Since each vertex is responsible for its parent edge, we will assume that the edges of T_3 are also stored in $A[\beta\,..\,\omega]$. Assign 1's to all *unmarked* edges *in* Q and assign 0's to the other edges in T_3. In the array, each marked edge $e \in Q$ (assigned a 0) finds the nearest unmarked edge $f \in Q$ (assigned a 1), if any, to its left. Then, $\mathrm{HEAD}''(e) = u$ if u is the child vertex of f. If there is no such f for a marked edge $e \in Q$, then $\mathrm{HEAD}''(e) = r'$, the root of T'. Finding the rightmost 1 in a boolean array of size k can be solved in $O(1)$ time using $O(k)$ processors in the CRCW PRAM [F]. So, we need $O(|T_3|^2)$ processors to compute $\mathrm{HEAD}''(e)$ for all marked edges $e \in Q$. This is where concurrent writes are required and a quadratic number of processors are used.

We, now, compute $\mathrm{HEAD}''(e)$ for $e \in T_3 \sim Q$. First, consider the newly marked edges. These are all adjacent to vertices in Q. So, $\mathrm{HEAD}''(e) = \mathrm{HEAD}''(f)$ where f, if any, is the marked edge in Q adjacent to e, and $\mathrm{HEAD}''(e) = r'$ if no such f exists. Second, consider the other marked edges e in T_3. These were marked in T' and are still marked. If $\mathrm{HEAD}'(e)$ is the child vertex of a newly marked edge f, then $\mathrm{HEAD}''(e) = \mathrm{HEAD}''(f)$.

Updating ND*'s is rather simple, so we omit it.*

Back to Step 2 of our overview in section 4.1: If all information is available, finding a basic path of T_i can be done in constant time using $O(|T_i|)$ processors. Linking $\mathrm{SUCC}(\cdot)$ can also be done in constant time. Note that the vertices in a basic path need not to be stored consecutively in an array.

This completes our proof of Lemma 2, and thus, of Theorem 4.

By Theorems 3 and 4, we have proved our main theorem.

Theorem 5. *For a tree T of degree three with n vertices, one can find a mincut linear arrangement of T in $O(\log n)$ time using $O(n^{2.5}/\log n)$ processors in the CRCW PRAM.*

References

[C] M.J. Chung *et al.*, Polynomial time algorithms for the min cut problem on degree restricted tree, *23rd FOCS*, pp. 262–271, 1982.

[D] J. Díaz, Graph layout problems, *17th MFCS*, LNCS, vol. 629, pages 14–23, 1992.

[D1] J. Díaz *et al.*, Efficient parallel algorithms for some tree layout problems, TR, LSI-92-33-R, UPC, Barcelona.

[F] F.E. Fich, P. Ragde, and A. Wigderson, Relations between concurrent-write models of parallel computation, *SIAM J. Computing*, **17** (1988) pp. 606–627.

[G] M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[J] J. JáJá, *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[Y] M. Yannakakis, A polynomial algorithm for the min-cut linear arrangement of trees, *J. ACM*, **32** (1985) pp. 950–988.