The Accelerated Centroid Decomposition Technique For Optimal Parallel Tree Evaluation In Logarithmic Time

by

Richard Cole†
Uzi Vishkin‡

Ultracomputer Note #108
Computer Science Department Technical Report #242
September, 1986

† Courant Institute. This research was supported in part by NSF grant DCR-84-01633 and by an IBM faculty development award.

‡ Courant Institute and Tel Aviv University. This research was supported in part by NSF grants NSF-DCR-8318874 and NSF-DCR-8413359, ONR grant N00014-85-K-0046 and by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-AC02-76ER03077.

ABSTRACT

A new general parallel algorithmic technique for computations on trees is presented. The new technique performs a reduction of the tree expression evaluation problem to list ranking; then, the list ranking provides a schedule for evaluating the tree operations. The technique needs logarithmic time using an optimal number of processors and has applications to other tree problems.

This new technique enables us to systematically order four basic ideas and techniques for parallel algorithms on tree: (1) The list ranking problem. (2) The Euler tour technique on trees. (3) The centroid decomposition technique. (4) The new *accelerated centroid decomposition (ACD)* technique.

1. Introduction

The model of parallel computation used in this paper is the concurrent-read exclusive-write (CREW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. A CREW PRAM allows concurrent access by several processors to the same common memory location for read but not for write purposes. See [V-83a] for a survey of results concerning PRAMs.

Let $Seq(n)$ be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form $O(Seq(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal*.

This paper deals with the general topic of parallel tree algorithms. It provides two kinds of contributions. One is methodological and the other is a concrete complexity result. Specifically, we suggest a methodological order for the fundamental parallel algorithmic techniques which relate to trees. This order identifies the list ranking problem as most basic. Second in this order is the Euler tour technique (of [TV-85] and [Vi-85]), which is based on list ranking. It provides a solution for many tree problems. Third is the centroid decomposition idea and the simple $O(\log^2 n)$ time parallel algorithm it implies. This idea was mentioned in [Me-83] among several others. So far nothing new has been added.

[MR-85] gave an elegant parallel algorithmic technique for a class of tree problems including evaluation of an arithmetic expression. Their contribution is not only to provide a new technique but also to characterize the class of problems that it can solve. Our new technique benefits greatly from their characterization.

We propose an alternative *accelerated centroid decomposition (ACD)* technique which solves the same class of problems using only the centroid decomposition idea and any appropriate parallel list ranking algorithm. Such a list ranking algorithm is used both indirectly, through the Euler tour technique, and directly. In other words, we solve this class of problems by a logarithmic time optimal parallel *reduction* into the list ranking problem. Since [CV-86b] provides a logarithmic time optimal parallel list ranking algorithm, the new accelerated centroid decomposition technique also achieves this desired efficiency! Recall that [MR-85] gave a deterministic logarithmic time parallel algorithm using a linear number of

processors and a randomized logarithmic time optimal parallel algorithm. Our technique is considerably simpler than this randomized algorithm. Another interesting difference with respect to the Miller-Reif technique is that they characterize their technique as a dynamic expression evaluation with no preprocessing. Our approach is to analyze the structure of the tree first. This is done efficiently using known methods. Later we use this analysis to schedule the evaluation part properly. Interestingly, the overall efficiency of our approach compares favorably with theirs. So it is ''evaluate on-the-fly'' versus ''analyze-first evaluate-later''.

To sum up. The incremental contribution of this paper is in presenting the fundamental parallel algorithmic techniques for trees in a ''textbook like'' fashion. Specifically, we show how to derive the more involved accelerated centroid decomposition technique from the more elementary techniques, namely the centroid decomposition technique, the Euler tour technique and list ranking. Also, as we mentioned above, the Euler tour technique is based on list ranking.

2. Preliminaries

The (extended) list ranking problem.

Input: n nodes each occupying an entry of an array whose size is n. Each node has at most one successor in the array. A node having a successor has the array index of this successor. Consider a path that starts from any node and follows the successor relation. We assume that such a path never provides a circuit. Each node v has a weight $w(v)$.

The problem: For each node compute the total weight of the nodes following it in its linked list.

Recently, [CV-86b] gave a logarithmic time parallel algorithm for this problem which uses an optimal number of processors $(n/\log n)$. The present paper relies on this algorithm. Note that for all practical purposes we could have used the $O(\log n \log^* n)$ time parallel algorithm of [CV-86a] which also uses an optimal number of processors and has the advantage that the constants which are hidden by the ''Big Oh'' notation are quite modest, by contrast with [CV-86b]; incidentally, it is easy to see that for $n \leq 2^{65000}$, $\log^* n \leq 5$ (see [AHU-74], p. 133).

The Euler tour technique on trees. Let $T=(V,E)$ be a rooted tree, where r is its root. For each node $v \neq r$, there is a single edge of the form $v \rightarrow u$; u is the *parent* of v.

Step 1. For each edge $u \rightarrow v$ in T add its anti-parallel edge $v \rightarrow u$. Denote the new graph H. Since $in-degree(v) = out-degree(v)$ for every node v, H has an Euler path that starts and ends at the root r. Step 2 computes this path into the vector of pointers D, where for each edge e of H, $D(e)$ will have the successor edge of e in the Euler path.

Step 2. For each node v of H we do the following. (Let $out-degree(v) = d$ in H and let the outgoing edges of v be $v \rightarrow u_1, ., v \rightarrow u_d$).

$D(u_i \rightarrow v) := v \rightarrow u_{i+1 \bmod d}$ for $1 \leq i \leq d$.

Now D has an Euler circuit. The ''correction'',

$D(u_d \rightarrow r) :=$ end of list (where $d = in-degree(r)$) gives an Euler path that starts and ends at r.

Next, we show how to use this Euler path in order to find for each node v the number of nodes in the subtree rooted at v, denoted SIZE(v).

Step 3.

Initialize:

for each e in H − T *pardo*

R(e) := 1

for each e in T *pardo*

R(e) := 0

In words: if e in H is the incoming edge of a node in the direction from the root then its weight is one and otherwise it is zero.

Apply a list ranking algorithm to find for each e in H its (weighted) distance to the end of the Euler path, storing the result in DISTANCE(e).

For each v≠r consider the edge e = v → u in T where u is the parent of v, and let $\bar{e}$ = u → v be the anti-parallel edge of e in H. It is not difficult to observe that,

SIZE(v) := DISTANCE($\bar{e}$) − DISTANCE(e) .

Also,

SIZE(r) := ''total weighted length of the Euler path'' + 1

Centroid decomposition

Let T=(V,E) be a rooted tree, where r is its root. Let n = |V| Recall that SIZE(v) is the number of nodes in the subtree of T rooted at v. Let the *centroid level* of v, denoted C−LEVEL(v), be $\lceil \log \text{SIZE}(v) \rceil$.*

Observation. Each node v has at most one child u such that C−LEVEL(u) = C−LEVEL(v).

Accordingly, we define the *centroid path* of v to be the longest directed path of nodes and tree edges, passing through v, where all the nodes on the path have the same centroid level as v. We note that there might be several disjoint centroid paths of the same centroid level.

This partition of the nodes of T into centroid paths is called the *centroid decomposition* of T. See Fig. 1. There are a few alternative definitions for the centroid decomposition notion. One such definition is given in Comment 2 in the next section.

[Me-83] considers problems on rooted trees for which there are serial algorithms that consist of moving from the leaves towards the root of the tree. It requires that these algorithms run fast in parallel, say O(log |V|) time, for *linear trees*, i.e., trees that consist of one simple path. Megiddo suggests considering the following $O(\log^2 |V|)$ time scheme for extending such linear tree algorithms into general trees:

for i := 0 to $\lceil \log n \rceil$ *pardo*

for each centroid path at level i *do*

* The base of all logarithms in this paper is two.

apply an adaptation of the linear tree algorithm

od

odpar

Note that nodes in centroid paths of level i are not being considered by this framework until the treatment of all centroid paths of lower levels is finished. In contrast, the new method works on nodes of centroid path i *in parallel* to working on nodes of lower centroid paths. Therefore, we call the new method the *accelerated centroid decomposition (ACD)* method.

3. The accelerated centroid decomposition method

Essentially, each step of the ACD method consists of applying, in parallel, several operations that reduce the tree at hand. Given an input tree, the method performs these tree reduction steps so that we end up with only the root. (Later we discuss the applicability of ACD for parallel algorithms that obey the above centroid decomposition scheme.)

The ACD method allows two kinds of operations (for simplicity, below, we consider only binary input trees): PRUNE - Node v becomes ready for the PRUNE operation as soon as it becomes a leaf. Applying PRUNE to v results in eliminating v from the tree. For some applications, it may be illegal to perform simultaneous PRUNE operations on both children of a given node; thus later comments explain how our solution can be made to avoid simultaneous PRUNEs. SHORTCUT - Let v be a node whose centroid level in the input tree is i and suppose that v is not an extreme node in its centroid path (i.e., its centroid path contains both its parent and one of its children, to be called its centroid child). Node v becomes ready for SHORTCUT after losing its non centroid child (the method implies that when this happens v must still have a child with the same centroid level as itself). Applying SHORTCUT to v results in eliminating v so that its parent becomes the parent of its child. It is *illegal* to perform simultaneously two SHORTCUT operations both at a node and at its child. RAKE and COMPRESS are the names that were used in the original paper of [MR-85] for operations, similar, though *not identical*, to PRUNE and SHORTCUT, respectively. This non-identicality was essential for our presentation and, therefore, we use different names.

The reader may find our description rather abstract. Therefore, it might be helpful to consider the following example throughout this presentation. Recall the *minimum vertex cover (MVC)* problem. Given a graph $G(V,E)$, a subset V' of vertices is a *vertex cover* of G if one endpoint of each edge in E is in V'. The MVC problem is to find a vertex cover whose cardinality is minimum. This problem is NP-complete for general graphs but has an obvious ‘‘leaf-to-root’’ linear time serial algorithm for rooted trees: Every leaf is not selected for the MVC. This implies that each vertex adjacent to a leaf is selected for the MVC. Next remove all the vertices that have been selected to be either inside or outside the MVC and their adjacent edges from the graph. We get one or more rooted trees and iterate this procedure for each of them. Nodes that become singletons are not selected for the MVC. It is easy to show that this vertex cover is indeed of minimum cardinality.

Next, we demonstrate PRUNE for the MVC problem. Consider a vertex v to which PRUNE should be applied. v is a leaf. If v has not been selected for the MVC then remove v and select its parent for the MVC. If v was selected then simply remove v.

We demonstrate SHORTCUT for the MVC problem. Let v be a node of some centroid path whose non centroid child x was removed. If x was placed in the MVC then so far we have not been obliged to add v to the MVC, while if x was not placed in the MVC then we are obliged to add v to the MVC. Thus there are three possibilities: 1. v is in the MVC. 2. Based on its descendents in its centroid path v is not in the MVC. Using the notion of motion from the leaves to the root and the idea of including the furthest vertex possible in the MVC we conclude (upon performing SHORTCUT at v) that v is not in the MVC. 3. Neither 1 nor 2 (namely, we do know yet whether v is in the MVC).

In order to understand the situation better let us be more specific. Consider an instance where the whole tree is one directed path (linear tree). Let u be the parent of v, and w be the (only) child of v. Then the ‘‘spirit’’ of the serial algorithm implies the following two dependencies: (a) u is in the MVC if and only if v is not, and (b) v is in the MVC if and only if w is not. This enables us to do the following SHORTCUT: remove v, ‘‘remember’’ dependency (b), and compose (a) and (b) into the following dependency: (c) u is in the MVC if and only if w is. Dependency (b) provides a pointer from v to w. Similarly, a later SHORTCUT may remove w and provide w with a pointer to a vertex which has not yet been removed and so on. This linked list of pointers starting from v must lead to a vertex whose membership in the MVC was determined after all SHORTCUTs and PRUNEs are over. Determining the membership of all vertices in the list is done by backtracking. We finish with this instance of linear trees by observing that a pair of dependencies of type (a) (opposite participation in MVC) or of type (c) (same participation in MVC) can be composed in a similar way to provide a dependency of one of these two forms.

We return to the MVC problem on general trees. As with linear trees we initially assign opposite participation dependency to any pair of successive vertices in the same centroid path, *ignoring* edges between vertices whose centroid level is different. Let u be the parent of v and w its child in the present tree and suppose that the centroid level of both u and w are the same as the centroid level of v. (We never perform SHORTCUT if this assumption does not hold). Also, recall that we assumed v no longer has a non centroid child. If possibility 3 holds we compose the dependencies between u and v and v and w into a dependency between u and w, thereby making w a child of u. We also remove v and ‘‘remember’’ the dependency between v and w in order to be able to determine later whether v is in the MVC. Suppose that either possibility 1 or 2 holds. In this case, our general method becomes redundant when it comes to the relation between u and w. It also makes w a child of u. However, this has no algorithmic meaning since there is no non-trivial dependency between w and u. (The reader should realize that the problem considered here is perhaps one of the easiest to be solved by the ACD method and therefore a few cases are degenerate and do not need the full power of the method). Finally, we consider direct implications on u. If we already know that u is in the MVC then we are in a degenerate case. Otherwise, using the dependency between u and v, we either decide that u is in the MVC or that: ‘‘based on its descendents in its centroid path u need not be in the MVC’’.

The new method actually has three stages: Preparatory stage, Scheduling stage and Evaluation stage.

The Preparatory Stage. The input for this stage is a rooted tree where each node has a linked list of its children (incoming edges) and its parent (outgoing edge). The following things are being computed:
1. The tree is ''binarized'': For each node that has more than two children replace its incoming edges by a binary tree where: 1. each child is a leaf in this binary tree; 2. the node is the root; and 3. if all leaves are pruned, the tree comprises a path. See Fig. 2. *Remark.* It is simple to adapt now the MVC implementation to this binarization. Each internal node in this binary tree has two kinds of children: an auxiliary node or leaves. The adaptation consists of the following initialization: 1. Assign opposite participation dependency to an internal node with respect to a leaf child. 2. Assign the same participation dependency to an internal node with respect to a child which is another internal node. It is not difficult to verify that this solves the MVC problem.
In general, the binarization may need some ad hoc effort which differs from one application to the next.
2. SIZE(v). SIZE(v) is the number of nodes in the tree rooted at v, for each node v.
3. TAIL(v), HEAD(v) and RANK(v). Consider the centroid path that contains v. TAIL(v) (resp. HEAD(v)) is the furthest descendent (resp. ancestor) of v on this path. RANK(v) is the length of the path from v to HEAD(v).

Let $v_1, v_2, .., v_{\alpha+1}$ be a centroid path at level i, where v_1 is the head of the path and $v_{\alpha+1}$ is the tail of the path. For reasons that will become clear later, we focus only on nodes $v_1, .., v_\alpha$. Let $SIZE_j$ be $SIZE(v_j) - SIZE(v_{j+1})$, $1 \leq j \leq \alpha$. In words, $SIZE_j$ is the size of the tree rooted at v_j excluding the tree rooted at v_{j+1}. Observe that $\sum_{j=1}^{\alpha} SIZE_j = SIZE(v_1) - SIZE(v_\alpha)$.
4. All α prefix sums $A_k = \sum_{j=1}^{k} SIZE_j$, for $1 \leq k \leq \alpha$. Thus $\sum_{j=1}^{\alpha} SIZE_j < 2^i$.

The Main (Scheduling) Stage. This stage is devoted to scheduling each node of the tree for either PRUNE or SHORTCUT. We provide an inductive description of the Main Stage. We will schedule the removal of each node of the tree and decide whether the removal is done by PRUNE or SHORTCUT. At the end we explain how to assign processors to the operations of each time unit. Our inductive description will make use of the following inductive claim regarding the reduced tree following time T(i), where i is the centroid level and T is some integer valued function.
The Inductive Claim.
A. The reduced tree at the end of time T(i) satisfies the following: 1. Every node, whose centroid level is $< i$, has been removed. 2. Every node, whose centroid level is i and is not the head of its centroid path, has been removed. 3. Every node, whose centroid level is i and is the head of its centroid path, is a leaf.
B. The reduced tree at the end of time $T(i) + 2$ satisfies the following: any centroid path, whose level is $i+1$, that contains at least two nodes (in the input tree) has two nodes which have not yet been removed. These two nodes are the tail and head of the centroid path.

Next, we assume the Inductive Claim for i and show how to satisfy it for $i+1$ within 4 time steps past time T(i). This clearly implies that $T(i) = 4i$. We will see that this also implies a total of O(log n)

time for the Evaluation Stage that follows and for the whole algorithm.

For $i=0$ the Inductive Claim holds trivially. Assume the Inductive Claim holds for i. We describe *iteration* $i+1$ of the Main Stage. It is devoted to satisfying the Inductive Claim for $i+1$. Satisfying item A of the Inductive Claim for $i+1$ is very simple: Recall that item B for i implies that after time $T(i) + 2$ each path of centroid level $i+1$ remains with at most two nodes. In case any of these two nodes has a child whose centroid level is $< i+1$ then such a child must be a leaf whose centroid level is i (by A3 for i).

Our solution for satisfying item A is as follows:
Schedule two serial steps of PRUNE: one at time $T(i) + 3$ and the second at time $T(i) + 4$ (as defined by Fig. 3).

The rest of this description is devoted to satisfying item B of the Inductive Claim for $i+1$. Consider the tree rooted at the head of the centroid path of level $i+2$ excluding the subtree rooted at the tail (See Fig. 1). Each node of the centroid path (recall that the tail is excluded in this part) has a *centroid child* and perhaps another, non-centroid, child. We would like to schedule the SHORTCUT operations to be applied to the nodes on the centroid path. However, the fact that different nodes of the centroid path become ready for SHORTCUT at different times complicates the situation.

Our algorithm uses: 1. The inductive assumption that if the tree rooted at any non-centroid child has x nodes then either this child has been removed or it is a leaf at time $T(\lceil \log x \rceil)$. 2. The fact, mentioned above, that the total number of descendants of the centroid path of level $i+2$ (excluding the tail) is $\leq 2^{i+1}$. 3. The data which were computed in the Preparatory Stage.

Let us focus on some centroid path at level $i+2$: $v_1, v_2, .., v_{\alpha+1}$. Since item B does not imply anything regarding the subtree rooted at $v_{\alpha+1}$, we focus further only on nodes $v_1, .., v_\alpha$. Let u_j be the non centroid child of v_j, for $1 \leq j \leq \alpha$. (Node v_j need not have a non centroid child.) Let $SIZE_j$ and their prefix sums A_j be as above. Recall that $\sum_{j=1}^{\alpha} SIZE_j \leq 2^{i+1}$.

By the inductive hypothesis, node u_j , $1 \leq j \leq \alpha$, becomes a leaf by time $t_j = T(\lceil \log SIZE(u_j) \rceil)$.

Our solution is:

PRUNE u_j at time step $t_j + 1$, for $j > 1$.
This makes node v_j ready for SHORTCUT.

Suppose $\alpha \geq 2$ (otherwise the inductive hypothesis B already holds). We will ensure that by the end of time $T(i+1) + 2$ the nodes $v_2, .., v_\alpha$ have all performed SHORTCUT. First, we give a concise description of our solution. Later, we use some illustrations in order to give a more intuitive explanation. For each j, $1 < j \leq \alpha$, consider the binary representation of $A_{j-1} - 1$ and $A_j - 1$. Let K_j be the index of the most significant bit in which these two binary representations are different.
(*Example*. Let the representations of $A_{j-1} - 1$ be 011011 and $A_j - 1$ be 010101. Then K_j is 4.)

Remarks:

1. For K_j to be well-defined, we concatenate zeros to the left of the binary representation of $A_{j-1} - 1$.
2. Bit K_j is zero in (the representation of) $A_{j-1} - 1$ and one in $A_j - 1$. (Since $A_j - 1$ is larger than $A_{j-1} - 1$).

Our solution is very simple to state:

SHORTCUT at node v_j, $1 < j \leq \alpha$, at time $T(K_j) + 2$.

In order to establish the validity of our solution we show:

Validity Claim 1. We never SHORTCUT at a node before it is ready.

To see this, observe that $SIZE_j = (A_j - 1) - (A_{j-1} - 1) \leq 2^{K_j} - 1$. By the inductive hypothesis, if node v_j has a non centroid child at time $T(K_j)$ then it must be a leaf and we PRUNE it at time $T(K_j) + 1$.

Validity Claim 2. We never perform two simultaneous SHORTCUTs at two adjacent nodes of the path. (That is, it is impossible that the two nodes are adjacent at the time at which the SHORTCUTs are performed. Earlier we called such SHORTCUTs illegal.)

To show this, we need the following observation:

Observation. For every $K_j = K_l$, where $1 < j < l \leq \alpha$, there exists $K_\beta > K_j$, where $j < \beta < l$. In other words, given two nodes v_j, v_l at which simultaneous SHORTCUTs are performed we are guaranteed to have node v_β between them at the time of these SHORTCUTs, and therefore they are legal and Validity Claim 2 follows.

Proof of Observation. Remark 2 implies that bit K_j at $A_j - 1$ is 1, and bit $K_j (= K_l)$ at $A_{l-1} - 1$ is 0. Therefore, for some $j < x < l$, bit K_j was 1 at $A_{x-1} - 1$ and 0 at $A_x - 1$. Apply Remark 2 again to conclude $K_x > K_j$, which proves the Observation.

For readers who seek a more intuitive explanation of our Scheduling Stage, we provide the following illustrative explanation. Consider an auxiliary balanced binary tree whose leaves are all binary strings of length $i+1$ ordered lexicographically. (Which is, clearly, also the order of the numbers that they represent). If $SIZE_j = 1$ for each j in the path then we could have simply SHORTCUT at each even location of the path and repeatedly done the same to the remaining path. Observe that this corresponds to the structure of the binary tree. More generally, we identify the half open interval of leaves $(A_{j-1} - 1 , A_j - 1]$ with node v_j, $1 < j \leq \alpha$. Denote by LCA_j the lowest common ancestor of leaf $A_{j-1} - 1$ and leaf $A_j - 1$ in the binary tree. The crucial observation is that K_j of our above solution is exactly the height of LCA_j in the tree (that is, the length of the path in the tree from LCA_j to its closest leaf). We leave it to the interested reader to verify that both Validity Claims above and the Completeness Claim below can be proved using the insight provided by this auxiliary binary tree.

Completeness Claim. Each node is scheduled for removal during the Main Stage.

Proof of Completeness Claim. Consider node v whose centroid level is $i+1$. If it is the tail of its centroid path then it will be removed using PRUNE at time $T(i) + 4$. If it is neither a tail nor a head

then it will be removed at time $T(j) + 2$ for some $j \leq i$, using SHORTCUT. If it is the head of its centroid path, there are two possibilities. Let u be the parent of v whose centroid level is j. If u is a head or tail then v is removed at time $T(j) + 3$, using PRUNE. Otherwise, v is removed at time $T(i) + 1$ using PRUNE.

Comment 1. If some application does not allow two simultaneous PRUNEs at two children of the same node we can arbitrarily schedule one before the other. Since this may happen only at a time step of the form $T(.) + 3$, this increases the number of time steps past time $T(i)$ until time $T(i+1)$ to 5 and, therefore, $T(i)$ to $5i$.

In the above presentation our goal was to get a time bound of the form $O(\log n)$ and we did not bother too much about the constant that multiplies $\log n$. The authors see a few ways for modifying the accelerated centroid decomposition method in order to reduce this constant. In the following comment we present one such way which seems to be of independent interest since it is based on an alternative notion of centroid decomposition. Our description focuses on the changes to the above scheduling method.

Comment 2. **Alternative centroid decomposition and Scheduling Stage**.
We partition the nodes of the tree into paths called *centroid lanes* (instead of centroid paths). Each internal node of the tree selects its child, whose SIZE is largest, to be its *subsequent* in its centroid path. (In the event of a tie one of the children is selected arbitrarily). It can be easily verified that: (1) each centroid lane comprises several centroid paths of decreasing levels (2) the tail of each centroid lane is always a leaf. (We see below that this simplifies considerably the situation described in Comment 1 above as we do not need to avoid performing simultaneous PRUNEs.) The *level* of a centroid lane is the centroid level of its head.

Later we describe an alternative scheduling stage. First, we mention two important differences with respect to the previous scheduling.
(1) We process tails of centroid lanes in the same way as other nodes of a lane. For this we extend the definition of K_j above to $K_{\alpha+1}$ in the natural way. Heads still get a special treatment. We note that whenever it is written below that SHORTCUT should be applied to a leaf in the reduced tree, then the convention is that PRUNE is actually applied.
(2) The total number of descendents of the centroid lane of level $i+1$ is $\leq 2^{i+1}$ (and not 2^i as for centroid paths).

The *Inductive Claim* is: at the end of time $T(i)$ each level i lane is reduced to its head node and at most one other node.

The Alternative Scheduling Method: Use three time steps per level.

Time step $T(i) + 1$: PRUNE level i nodes that are not head nodes.
Time step $T(i) + 2$: PRUNE level i head nodes.
Time step $T(i) + 3$: SHORTCUT at each node v_j for which K_j equals i. (Note that apart from the head node there is just one node in each level $i+1$ lane that is not SHORTCUT by this step.)

This finishes Comment 2.

The Evaluation Stage.

The only remaining problem for actual performance of PRUNE and SHORTCUT as per the schedule found in the previous stage is that of assigning processors to jobs. Each node knows its schedule for removal. This schedule is a positive number which is $O(\log n)$. We sort the nodes according to their schedules. Lemma 3.4 in [R-85] gives a logarithmic time optimal parallel bucket sort *deterministic* algorithm (which is also quoted in [CV-86a]) for sorting n positive numbers which are $\leq \log n$. It is easy to extend this for positive numbers which are $O(\log n)$ without changing the time and processor complexities. We use this algorithm to sort the schedules. The assignment of any number of $p \leq n/\log n$ processors to nodes is easy. At each time of the implementation perform the next p jobs that should be performed in the time unit of our schedule which is currently being implemented. If there are less than p remaining jobs perform all p of them; this leaves some of the processors idle for this time step of the implementation (see [CV-86a] for details of a similar assignment of processors to jobs).

Complexity. We have shown that, using the logarithmic time optimal parallel list ranking algorithm, our ACD method has the same efficiency. Namely, it takes logarithmic time using an optimal number of processors.

Remarks.

1. [MR-85] demonstrate their technique for the problem of expression evaluation. Given an arithmetic expression, the idea is to apply the logarithmic time optimal parallel algorithm of [BV-85] in order to transform it into a computation tree form. Given a computation tree, they show how to apply their method in order to evaluate it. Note, however, that in [MR-85] a node is ready for SHORTCUT only if both the node and its parent do not have non-centroid children. It is easy to modify our algorithm to handle such a change in the definition of SHORTCUT. Simply, let another variable S_j play the role of $SIZE_j$ for each j. Define $S_j = \max(SIZE_j, SIZE_{j-1})$. We then compute prefix sums with respect to S_j and finish as before, but with the following exception. Note that one K_j at a level $i+2$ path may be $i+2$. We add a time step between time step $T(i) + 2$ and time step $T(i) + 3$ in which we SHORTCUT at each node v_j of level $i+2$ for which $K_j = i+2$. It is easy to see that this increases the time that elapses between $T(i)$ and $T(i+1)$ by one time unit. Thus the total running time is still $O(\log n)$ using an optimal number of processors.

2. [GR-86] showed how to get a logarithmic time optimal parallel algorithm for the problem of expression evaluation for expressions which are given in the form needed for the parsing algorithm of [BV-85]. Their algorithm works in three stages. The first stage simply applies the algorithm of [BV-85]. The second stage, which is the new part, is a nice reduction of the evaluation problem into a smaller instance of the evaluation problem. The third stage consists of applying the method of [MR-85] to this smaller instance of the evaluation problem. We believe that our method compares favorably with theirs because of the following three differences. (a) Their improvement seems not to carry through when the computation tree is already given. (b) Therefore, it does not generalize to other tree problems to which the method of [MR-85] is applicable. (c) It essentially applies the method of [MR-85] to a smaller problem instance,

while the introduction explained why our method and presentation are methodologically interesting.

3. Given a graph we define a subset of its vertices to be a dominating set if each vertex outside this subset is adjacent to a vertex in the dominating set. We note that there is a serial algorithm for finding minimum dominating sets on trees; this algorithm is similar to the serial algorithm for the MVC problem. It is straightforward to apply the ACD method to find the minimum dominating set for a tree. We also note that a very recent paper [H-86], suggested applying the Miller-Reif method to several more problems on trees. The ACD method is applicable to all these problems, as well.

4. Apparently there are ‘‘leaf-to-root’’ serial algorithms for which the SHORTCUT operation cannot be adapted. For instance, consider the simple serial algorithm for the following tree partitioning problem. The algorithm is given in [Me-83]. Given a tree T, where a nonnegative weight is associated with each vertex, and given a nonnegative number λ, the problem is to delete the maximal number of edges of T so that the weight of each resulting subtree is at least λ. We note that we run into similar difficulties in trying to adapt the COMPRESS operation of [MR-85] to the serial algorithm for this problem.

4. Acknowledgement. Helpful comments by Zvi Kedem, Yael Maon and Baruch Schieber are gratefully acknowledged.

5. References

[AHU-74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[BV-85] I. Bar-On and U. Vishkin, Optimal parallel generation of a computation tree form, ACM Trans. on prog. Lang. and Sys. 7,2 (1985), 348-357.

[CV-86a] R. Cole and U. Vishkin, Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms, Proc. 18th ACM Symp. on Theory of Computing, 1986, 206-219, accepted Inf. and Cont.

[CV-86b] R. Cole and U. Vishkin, Approximate and exact parallel scheduling with applications to list, tree and graph problems, Proc. 27th Symp. on Foundations of Computer Science, 1986, to appear.

[GR-86] A. Gibbons and W. Rytter, An optimal parallel algorithm for dynamic tree expression evaluation and its applications, Research Report 77, Dept. of Computer Science, Univ. of Warwick, Coventry, CV4 7AL, England, 1986.

[H-86] X. He, The general tree algebraic computations and its applications in parallel algorithms design, preprint, 1986, Dept. of Computer and Information Science, Ohio State University, Columbus, OH 43210.

[Me-83] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, JACM 30(1983), 852-865.

[MR-85] G.L. Miller and J.H. Reif, Parallel tree contraction and its applications, Proc. 26th Symp. on Foundations of Computer Science, 1985, 478-489.

[R-85] J.H. Reif, On optimal parallel algorithm for integer sorting, Proc. 26th Symp. on Foundations of Computer Science, 1985, 496-503.

[TV-85] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, SIAM J. Comput. 14,4 (1985), 862-874.

[V-85] U. Vishkin, On efficient parallel strong orientation, Information Processing Letters 20 (1985), 235-240.