Optimal Parallel Evaluation of Tree-Structured Computations
by Raking (Extended Abstract)

S. Rao Kosaraju* Arthur L. Delcher
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218

Abstract. We show that any arithmetic expression of size $n$ can be evaluated on an EREW PRAM with $O(n/\log n)$ processors in $O(\log n)$ steps. A major contribution is the simplicity of the algorithm. In contrast with existing algorithms which require independent RAKE and COMPRESS operations, our algorithm combines the RAKE and COMPRESS into one simple operation. In fact, our algorithm can be viewed as avoiding COMPRESSes entirely and simply performing RAKES. The algorithm can be modified easily to evaluate every subexpression of the original arithmetic expression. We also show how it can be applied to the following problem: Given a positive constant $\lambda$ and a tree with weighted nodes, partition the tree into minimal components subject to the constraint that the sum of the node weights in each component is at least $\lambda$.

1. Introduction

Arithmetic expression evaluation has been established as an important paradigm for solving several problems. Miller & Reif [6] and Rytter [8] independently gave exclusive-read exclusive-write (EREW) parallel random access machine (PRAM) algorithms for evaluating any arithmetic expression of size $n$, which run in $O(\log n)$ time using $O(n)$ processors. Subsequently Cole & Vishkin [4] and Gibbons & Rytter [5] independently developed $O(\log n)$-time, $O(n/\log n)$-processor EREW PRAM algorithms. In the next section we give an extremely simple $O(\log n)$-time, $O(n/\log n)$-processor EREW PRAM algorithm for the same problem. We view our algorithm as a dramatically simplified version of the algorithm in [5]. We also show how the algorithm can be modified to yield the values of all the subexpressions in $O(\log n)$ time using an $O(n/\log n)$-processor EREW PRAM.

In section 3, we apply the technique to obtain an $O(n/\log n)$-processor, $O(\log n)$-time EREW PRAM algorithm for the following problem: Given a positive constant $\lambda$ and a tree $T$ with non-negative weights assigned to its nodes, delete a maximal set of edges from $T$, such that the total weight of the nodes in each resulting component is at least $\lambda$.

2. Main Algorithm

For simplicity of presentation we assume that the expression to be evaluated consists of just constants and the operations of addition and multiplication. The algorithm can be modified easily to accommodate subtraction and division, and other non-arithmetic

*Supported by the National Science Foundation under Grant DCR-8506361
operations. We can even drop the commutativity requirement for the + and \times operations. Again, for simplicity, we assume that this expression is given in the form of a binary tree in which each leaf has an associated constant value, and each internal node has an associated operator, + or \times. Each internal node in this tree has exactly 2 children—a property of binary trees we shall refer to as properness. For any node \( u \) in this tree, let \( \text{Par} (u) \), \( \text{Gpar} (u) \), \( \text{Sib} (u) \), \( \text{Left} (u) \) and \( \text{Right} (u) \) denote its parent, grandparent, sibling, left child and right child respectively. We call \( u \) a left node (resp. right node) if \( u \) is the left child (resp. right child) of \( \text{Par} (u) \). We measure the size of the expression as the number of its leaves.

Our goal is to compute the value of the entire expression, i.e., the value of the root.

To motivate the reader, we first give an informal description of our algorithm.

A. Informal Description

We evaluate the expression by reducing the size of its tree until only the root is left. The operation we employ to remove nodes from the tree is called RAKE because it is applied only to leaf nodes. (\cite{7} describes a similar operation, called shunt.) When we RAKE a leaf \( w \), we remove \( w \) and \( \text{Par} (w) \) from the tree, and make \( \text{Sib} (w) \) the child of \( \text{Gpar} (w) \) in the position vacated by \( \text{Par} (w) \). Notice that this operation preserves properness of binary trees. To prevent modifying the value of the root as nodes are removed from the tree, we adjust the information stored at \( \text{Sib} (w) \).

Suppose that at some stage the leaves of the tree in left-to-right order are \( w_1, w_2, \ldots, w_m \). In the next phase we will RAKE, in parallel, all the odd-subscripted leaves \( w_1, w_3, w_5, \ldots \) by an \( O (m) \)-processor EREW PRAM in \( O (1) \) steps. The resulting proper binary tree will have \( m/2 \) leaves. Thus starting with an \( n \)-leaf proper binary tree, we need to perform \( \log_2 n \) phases before the tree is reduced to a single node. Note that the total number of steps performed by the complete algorithm is \( O (n) \). Hence, by Brent’s Theorem, we have an \( O (n/\log n) \)-processor EREW PRAM algorithm to evaluate the given tree in \( O (\log n) \) steps. By performing the phases in reverse, we can compute the value of each subexpression within the same processor and time bounds.

We now make the above description more precise.

B. Basic Algorithm

At any intermediate stage we maintain a proper binary tree with a constant \( c_w \) at each leaf \( w \), and an operator, \( \circ_u \) (+ or \times), at each internal node \( u \). Every node \( u \) stores pointers to its parent, left child and right child, as well as a bit, \( \text{Side} (u) \), indicating whether it is the left or right child of its parent. Each node \( u \) also stores a pair of numbers \( (a_u, b_u) \) which represent the linear expression \( a_u X + b_u \) where \( X \) is an indeterminate. For simplicity, we shall say that \( u \) stores the linear expression \( a_u X + b_u \).

We now recursively define the value at a node in the tree. The value at a leaf \( w \) is its associated constant \( c_w \). For any internal node \( u \) with operator \( \circ_u \), if the values at its left and right children are \( \text{Val}_L \) and \( \text{Val}_R \), and the expressions at the left and right children are \( a_L X + b_L \) and \( a_R X + b_R \), respectively, then the value at \( u \) is defined to be the result of the expression

\[
(a_u \circ_u (a_l \text{Val}_L + b_L)) \circ_u (a_R \text{Val}_R + b_R).
\]

Note that the value at node \( u \) does not involve the expression \( a_u X + b_u \); this expression