Tree-Based Parallel Algorithm Design[*]

Gary L. Miller[†] Shang-Hua Teng[‡]

Abstract

In this paper, a systematic method for the design of efficient parallel algorithms for the dynamic evaluation of computation trees and/or expressions is presented. This method involves the use of uniform closure properties of certain classes of unary functions. Using this method, optimal parallel algorithms are given for many computation tree problems which are important in parallel algebraic and numerical computation, and parallel code generation on **E**xclusive **R**ead, **E**xclusive **W**rite (EREW) **P**arallel **R**andom **A**ccess **M**achines (PRAMs). New techniques for prudent evaluation, dynamic processor scheduling, and load balancing are also provided.

[*]The preliminary version of this paper appeared in the *Second International Conference on Supercomputing*, Santa Clara, May 1987, pages 392–403.

[†]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. Supported in part by National Science Foundation grant CCR-9016641. Part of the work was while the author was at the Department of Computer Science, University of Southern California, Los Angeles, CA 90089-0782.

[‡]Department of Computer Science, University of Minnesota, Minneapolis, MN 55455. Supported by an NSF CAREER award. Part of the work was done while the author was at the Department of Computer Science, University of Southern California, Los Angeles, CA 90089-0782.

1 Introduction

Computation involving trees plays an important role in many computational problems, such as tree isomorphism testing, dynamic expression evaluation [17, 11], parallel optimal arithmetic code generation [7], parallel message decoding [22], connected component finding [9], graph planarity testing [15] and other graph problems [12, 13].

Miller and Reif [17] presented a general technique, *parallel tree contraction*, for developing processor-efficient parallel graph algorithms. They gave the first parallel tree contraction algorithm which takes $O(\log n)$ time, using n processors deterministically, or $n/\log n$ processors on randomized Concurrent Read, Concurrent Write (CRCW) Parallel Random Access Machines (PRAMs). Using new tree-partition, list-ranking, and isolated-compress techniques, Gazit, Miller, and Teng [10] presented an optimal, $O(\log n)$ time, $n/\log n$ processor, deterministic parallel algorithm for parallel tree contraction on Exclusive Read and Exclusive Write (EREW) PRAMs. As a consequence of the results in [17, 10], the dynamic evaluation of arithmetic expressions can be done in $O(\log n)$ time, using $O(n/\log n)$ processors on an EREW PRAM, deterministically. Independently, optimal parallel tree contraction algorithms have been developed in [1, 16].

However, the parallel complexity of several classes of computation trees, such as trees or expressions over $\{\mathbb{R}, \min, \max, +, -, \times, \div\}$, is still unknown. The applicability of the parallel tree contraction technique to the computation tree problem remains to be an interesting issue.

In this paper, some *algebraic closure properties* of certain classes of unary functions will be studied, and their importance in the development of efficient tree-based parallel algorithms will be described. We present a systematic method for constructing efficient parallel algorithms for the dynamic evaluation of computation trees and/or expressions. This method involves the use of uniform closure properties of certain classes of unary functions. Using this method, *optimal* parallel algorithms are given for many computation tree problems which are important in parallel algebraic and numerical computation, and parallel code generation on EREW PRAMs. In particular,

- Trees over $\{\min, \max, +, \times\}$ of size n can be evaluated in $O(\log n)$ time, using $n/\log n$ processors on EREW PRAMs.
- Trees over $\{\mathbb{R}^+, \min, \max, +, \times, \div\}$ of size n can be evaluated in $O(\log n)$ time, using $n/\log n$ processors on EREW PRAMs.
- Trees over $\{\mathbb{R}, \min, \max, +, -, \times, \div\}$ can be evaluated in $O(\log^2 n)$ time, using n processors on EREW PRAMs. The design of this parallel algorithm calls for some new techniques for prudent evaluation, dynamic processor scheduling, and load balancing.
- An optimal code of an arithmetic expression, given in tree form of size n, can be constructed in $O(\log n)$ time, using $n/\log n$ processors on EREW PRAMs.

One important issue in our method is to construct a set of unary functions over a given algebraic system, which *minimumly* satisfies the algebraic closure properties required, and to design data structure suitable for parallel processing to represent these unary functions. Systematic methods are developed for function representation, closure properties proof, and minimality proof for several important computation tree problems.

The model of parallel computation used in this paper is the EREW PRAM [6, 24]. All our algorithms can be implemented randomly on many realistic message-passing based parallel machines, such as hypercube-based parallel machine, butterfly network, with at most a factor of $O(\log n)$ increasing of run time. [20, 23, 14].

2 Preliminaries

An algebra is a 2–tuple $\mathcal{A} = \{\mathcal{D}_\mathcal{A}, \mathcal{OP}_\mathcal{A}\}$ where $\mathcal{D}_\mathcal{A}$ is a set of constants (finite or infinite) called the *domain* and $\mathcal{OP}$ is a finite set of operators on $\mathcal{D}$. In this paper, all algebras are assumed to be closed. A *tree* used in this paper is a rooted and ordered tree with pointers from child to parent.

An *algebra tree* is a 2-tuple $\{\mathcal{A}, \mathcal{T}\}$, where $\mathcal{T}$ is a tree where each leaf is labeled with a value from $\mathcal{D}_\mathcal{A}$, each internal node is labeled with an operator from $\mathcal{OP}_\mathcal{A}$.

The *value* of a node v, denoted by $value(v)$, is defined inductively:

- If v is a leaf, then $value(v)$ is equal to the label of v.
- If v is an internal node labeled $\odot$, with children $w_1, \ldots, w_l$, then:

$$value(v) = \odot(value(w_1), ..., value(w_l)),$$

The *computation tree problem* is defined to be the problem of computing the values of all nodes in a given computation tree.

2.1 Unary Functions and Closure Properties

Any function of form $f : \mathcal{D} \rightarrow \mathcal{D}$ is called a *unary function* over the domain $\mathcal{D}$.

In [18], the authors proposed to use the uniform closure properties of certain unary function classes to design efficient parallel algorithms for the circuit value problem. In this paper, we extend this method to computation tree problems with more complex domains. The following two closure properties are discovered to be important in the systematic design of optimal parallel algorithms for tree-based problems.

Definition 2.1 (Composition) *A unary function class $\mathcal{F}$ is* **closed under composition** *if for all $f_1, f_2 \in \mathcal{F}$, $f_2 \circ f_1 \in \mathcal{F}$.*

Definition 2.2 (Projection) *A unary function class $\mathcal{F}$ is* **closed under projection** *over an algebra $\mathcal{A}$, if for all $\odot \in \mathcal{OP}_\mathcal{A}$, for all $a_1, \ldots, a_l \in \mathcal{D}_\mathcal{A}$, and for all $i : 1 \leq i \leq l$:*

$$\odot(a_1, ..., a_{i-1}, x, a_{i+1}, ..., a_l) \in \mathcal{F},$$

where l is the number of places of $\odot$.

Note that composition operation is associative. Hence, if $\mathcal{F}$ is closed under composition, then $\{\mathcal{F}, \circ\}$ forms a *semi-group.*

A *function tree* over an algebra $\mathcal{A}$ is a labeled tree where one leaf is labeled with a variable and all other leaves are labeled with values from $\mathcal{D}_{\mathcal{A}}$; each internal node is labeled with an operator from $\mathcal{OP}_{\mathcal{A}}$. Each function tree defined a unary function in the natural way.

A unary function f is a *tree function* over an algebra $\mathcal{A}$, if there is a function tree over $\mathcal{A}$ defines f.

A unary function class $\mathcal{F}$ is *minimumly closed* over an algebra $\mathcal{A}$, if $\mathcal{F}$ is closed under composition and projection over $\mathcal{A}$, and no proper subset of $\mathcal{F}$ has those properties.

Example 2.3 *Let $\mathcal{F}$ be the class of linear fractions over $\mathbb{R}$, the set of real numbers; then, $\mathcal{F}$ is minimumly closed over $\{+,-,\times,\div\}$.*

Proof: $\mathcal{F}$ is closed under composition, because for all $f_1(x) = \frac{a_1x+b_1}{c_1x+d_1} \in \mathcal{F}$ and $f_2(x) = \frac{a_2x+b_2}{c_2x+d_2} \in \mathcal{F}$

$$f_2 \circ f_1(x) = f_2(f_1(x)) = \frac{a_3x+b_3}{c_3x+d_3} \in \mathcal{F},$$

where $a_3 = a_1a_2 + b_3c_1, b_3 = a_2b_1 + b_2d_1, nc_3 = a_1c_2 + c_1d_2, d_3 = b_1c_2 + d_1d_2$.

$\mathcal{F}$ is closed under projection over $\{+,-,\times,\div\}$, because $f_{+a}(x) = a+x \in \mathcal{F}$, $f_{-a}(x) = a-x \in \mathcal{F}$, $f_{\times a}(x) = ax \in \mathcal{F}$, $f_{\div a}(x) = a/x \in \mathcal{F}$.

$\mathcal{F}$ is minimumly closed over $\{+,-,\times,\div\}$, since for all $a,b,c,d \in \mathcal{D}$:

$$\frac{ax+b}{cx+d} = \frac{a}{c} + \frac{bc-ad}{c^2x+cd}$$

The closure properties of linear fraction have been used in several papers, including [3, 17]. This example is used here for the purpose of illustration.

Let $\mathcal{UTF}_{\mathcal{A}}$ denote all tree functions over $\mathcal{A}$. The following proposition follows directly from the definition of minimumly closed unary function classes.

Proposition 2.4

1. *$\mathcal{UTF}_{\mathcal{A}}$ is* **minimumly closed** *over $\mathcal{A}$.*
2. *A unary function class $\mathcal{F}$ is minimumly closed over an algebra $\mathcal{A}$ iff $\{\mathcal{F}, \circ\}$ is a semi-group and $\mathcal{F}$ is closed under projection over $\mathcal{A}$, and $\mathcal{F} \subseteq \mathcal{UTF}_{\mathcal{A}}$.*

The *size* of a function tree is the number of nodes in the tree. The *tree size* of a tree-function f, is defined to be the minimum size of the function tree that defines f.

Definition 2.5 (Uniform Closure Properties) *$\mathcal{F}$ is (T,P)-uniform closed over $\mathcal{A}$ if $\mathcal{F}$ is closed under composition and projection over $\mathcal{A}$; and for all $f \in \mathcal{F}$ of tree size n, f can be evaluated in $O(T(n))$ time, using $P(n)$ processors; and the composition of two functions f and $g \in \mathcal{F}$ of tree size no more than n can be computed in $O(T(n))$ time, using $P(n)$ processors, where $T(n)$ and $P(n)$ are two integer functions.*

3 Dynamic Parallel Tree Evaluation

The main idea behind the dynamic parallel-tree-evaluation algorithm is to *partially* evaluate subtrees with only one unknown leaf and compress the information into a succinct form. The algorithm dynamically partitions the tree into independent subtrees, each of which has only one unknown leaf, and the set of subtrees are balancely scheduled over parallel processors which compute the compressed representations of the unary functions defined by the subtrees.

In this section, we present a generic parallel tree evaluation algorithm which simulates an improved version of the *parallel tree contraction* of Miller and Reif [10, 17]. This algorithm serves as a template for the systematic design of many tree based parallel algorithms. The *parallel-tree-ontraction* technique is used to efficiently generate independent tasks and dynamicly schedule parallel processors. We will prove the following Generic Theorem.

Theorem 3.1 (Generic Theorem) *Let $\mathcal{T}$ be a tree of size n over an algebra $\mathcal{A}$. If there exist a unary function class $\mathcal{F}$ which is (T, P)-uniform closed over $\mathcal{A}$, then $\mathcal{T}$ can be evaluated in $O(T(n)\log n)$ time, using $P(n)n/\log n$ processors, deterministically, on an EREW PRAM.*

3.1 Tree Partition and Parallel Tree Contraction

Tree contraction is a bottom-up procedure that reduces a tree to its root. It applies two abstracted operations, RAKE and COMPRESS, in a pipeline way, where RAKE removes all of the leaves in the tree and COMPRESS halves each chain by pointer–jumping.

Miller and Reif [17] presented the first parallel tree contraction algorithm, which takes $O(\log n)$ time using n processors, deterministically, or $n/\log n$ processors for a randomized algorithm on a CRCW PRAM. Gazit, Miller, and Teng [10] introduced two techniques to optimize the parallel tree contraction algorithm and implemented it on an EREW PRAM.

Their algorithm has two stages. In the first stage, the given tree is partitioned into a collection of $O(n/\log n)$ subtrees, called $\log n$-*bridges*, each contains at most one leaf, called a *leaf-variable*–that is, an internal node in the original tree. Moreover, the size of each bridge is bounded by $O(\log n)$ (see Figure 1). They present an algorithm to find the partition in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM.

There are two kinds of bridges: those which contain no leaf variables, called *leaf-bridges*; and those which contain one leaf variable, called an *edge-bridge*. Hence, one processor can be assigned to each bridge to evaluate it in $O(\log n)$ time. The evaluation of a leaf-bridge outputs the value of all nodes in the subtree. The evaluation of an edge-bridge, in which r_s is the root and l_s is the leaf-variable, outputs a unary function f_s such that $value(r_s) = f(value(l_s))$. Note that each leaf-variable is the root of some bridge. A tree T' can be defined over all leaf-variables and the root of the original tree, such that if the value of each node in T' is known, then the value of all nodes in the original tree can be computed in $O(\log n)$ time, using $n/\log n$ processors (See [10] for details). Therefore, using $O(\log n)$ time and $n/\log n$ processors, a computation tree problem of size n can be reduced to one of size $O(n/\log n)$.

The second stage applies a technique called ISOLATED COMPRESS that contracted a tree of size m to its root in $O(\log m)$ time, using m processors on an EREW PRAM. Two implementations

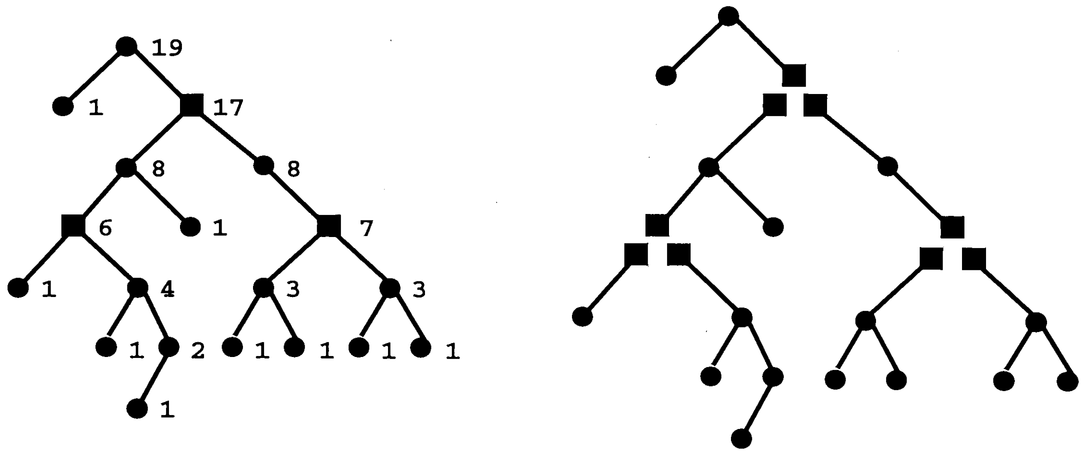

Figure 1: Example of a Tree Partition

of ISOLATED COMPRESS (see appendix A) was given in [10]. Those procedures will be discussed later in this paper.

The following is an algorithm for the second stage.

```
ALGORITHM The EREW Tree-Contraction
 begin
    while $V \neq \{r\}$ do
          in parallel for all $v \in V - \{r\}$ do
          if $v$ is a leaf, mark its parent and remove it;               (RAKE)
          isolate all the chains of the tree;                       (ISOLATION)
          COMPRESS each chain in isolation;                   (LOCAL COMPRESS)
          if a chain is a single vertex then unisolate it.       (INTEGRATION)
 end
```

Theorem 3.2 (Tree Contraction[10]) *Tree contraction can be performed deterministically in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM.*

3.2 Basic Operations and the General Algorithm

The purpose of the general parallel tree evaluation algorithm is to simulate the above EREW parallel contraction algorithm. The feasibility of the simulation is guaranteed by the closure properties of the unary function class. Some basic operations are defined below. For simplicity, the value of a node v is denoted by v itself. Let $\odot_v$ denote the operator of v.

Our basic technique is to introduce a unary function for each tree edge. These edge functions are updated in the process of parallel tree contraction to ensure that the value of each tree node is invariant during the application of parallel tree contraction. Let f_i denote the unary function of edge (v_i, v). Initially, all edges of the tree are associated with the identity function.

```
Procedure TRIMMING
    for all nodes $v$ in the tree whose children $v_1, \ldots, v_k$ are leaves, do
        $v \leftarrow \odot_v(f_1(v_1), \ldots, f_k(v_k))$;
    delete edge $(v_i, v)$ $(1 \leq i \leq k)$.

Procedure PROJECTING
    for all nodes $v$ with children $v_1, \ldots, v_l$ in which all but $u_i$ are leaves, do
        1) compute $f \in \mathcal{F}$, such that $f(x) = \odot_v(f_1(v_1), ..., f_{i-1}(v_{i-1}), x, ..., f_l(v_l))$;
        2) for all $j \neq i$ do
            delete edge $(v_j, v)$;
            label $(v_i, v)$ by $f \odot f_i$.

Procedure LOCAL COMPRESS
    use ISOLATED-COMPRESS or PRUDENT-COMPRESS in Appendix A.
```

The choice of which compress procedures to use depends upon the the uniformity of the unary function class, which will be discussed later in Section 5. The above Local Compress procedure is supported by the composition closure property.

The parallel tree evaluation algorithm can be specified as:

```
ALGORITHM Parallel Tree Evaluation
  begin
    while $V \neq \{r\}$ do
      in parallel do
        TRIMMING; PROJECTING;                                   (RAKE)
        isolate all the chains of the tree;                     (ISOLATION)
        LOCAL COMPRESS;                                         (LOCAL COMPRESS)
        if a chain is a single vertex, then unisolate it;       (INTEGRATION)
      run the dynamic expansion procedure given in [10].
  end
```

Lemma 3.3 *At the end of the algorithm, the value associated with each node is the value of the node.*

Proof: The lemma follows from a direct inductive argument on the number of applications of the basic operations, the definition of the composition and projection properties, and the correctness proof of the expansion procedure of [17] and [10]. □

Since *TRIMMING* together with *PROJECTING* simulate *RAKE* in $T(n)$ time, use $P(n)$ processors, *ISOLATION* and *LOCAL COMPRESS* are defined in the same way as those in [10]. It follows from Theorem 3.2 that Parallel Tree Evaluation takes $O(T(n) \log n)$ time, using $P(n)n/\log n$ processors on an EREW PRAM. This concludes the proof of Theorem 3.1.

4 A Reduction Lemma

Theorem 3.1 demonstrates the relationship between parallel tree evaluation and all parameters of algebra trees. The constraint that the computation of composition and projection must be bounded by small $T(n)$ and $P(n)$ shows that the class of unary functions need be not only *closed*, but also *uniform*. Thus, it is important to develop a systematic method for unary function construction, closure property proof and minimality proof.

Let $\mathcal{D} \subseteq \mathbb{R}$. For all a and $b \in \mathcal{D}$, let $B_{a,b}(x) = \min(a, \max(x, b))$; and let $\mathcal{B}_\mathcal{D} = \{B_{a,b}(x) \mid a, b \in \mathcal{D}\}$. Clearly, $\mathcal{B}_\mathcal{D}$ is minimumly closed under composition, and projection over $\{\min, \max\}$, i.e., $\{\mathcal{B}_\mathcal{D}, \circ\}$ is a semi-group.

Throughtout this section, let $\mathcal{F}$ be a set of unary function over $\mathcal{D}$, which contains the identity function. $\mathcal{F}$ is monotonic if all functions from $\mathcal{F}$ are monotonic over $\mathcal{D}$. Let $\mathcal{B}_\mathcal{D} \circ \mathcal{F} = \{b \circ f \mid b \in \mathcal{B}_\mathcal{D}, f \in \mathcal{F}\}$, and let $\mathcal{F} \circ \mathcal{B}_\mathcal{D} = \{f \circ b \mid b \in \mathcal{B}_\mathcal{D}, f \in \mathcal{F}\}$.

Lemma 4.1 *If $\mathcal{F}$ is monotonic over a domain $\mathcal{D}$, then $\mathcal{B}_\mathcal{D} \circ \mathcal{F} = \mathcal{F} \circ \mathcal{B}_\mathcal{D}$.*

Proof: It suffices to show that for all $B_{a,b} \in \mathcal{B}_\mathcal{D}$, and for all $f \in \mathcal{F}$, there exists $B_{a',b'} \in \mathcal{B}_\mathcal{D}$ such that $f \circ B_{a,b} = B_{a',b'} \circ f$.

Note that if f is monotonically increasing, then $f \circ B_{a,b} = B_{f(a),f(b)} \circ f$. While if f is monotonically decreasing, then $f \circ B_{a,b} = B_{\max(f(a),f(b)),f(a)} \circ f$. Hence, the lemma follows. □

Lemma 4.2 (Reduction Lemma) *If $\mathcal{F}$ is monotonic over domain $\mathcal{D}$ and minimumly closed over an operator set $\mathcal{OP}$, then $\mathcal{B}_{\mathcal{D}\cup\{\pm\infty\}} \circ \mathcal{F}$ is minimumly closed over $\mathcal{OP} \cup \{\min, \max\}$.*

Proof: For simplicity, let $\mathcal{B} = \mathcal{B}_{\mathcal{D}\cup\{\pm\infty\}}$.

To prove $\mathcal{B} \circ \mathcal{F}$ is closed under composition, it suffices to show that $\mathcal{B} \circ \mathcal{F} \circ \mathcal{B} \circ \mathcal{F} = \mathcal{B} \circ \mathcal{F}$.

It follows from Lemma 4.1 and the assumption that $\mathcal{F}$ is monotonic that $\mathcal{B} \circ \mathcal{F} = \mathcal{F} \circ \mathcal{B}$, hence,

$$\mathcal{B} \circ \mathcal{F} \circ \mathcal{B} \circ \mathcal{F} = \mathcal{B} \circ \mathcal{B} \circ \mathcal{F} \circ \mathcal{F}$$

Since both $\mathcal{B}$ and $\mathcal{F}$ are closed under composition, it follows that $\mathcal{B} \circ \mathcal{F} \circ \mathcal{B} \circ \mathcal{F} = \mathcal{B} \circ \mathcal{F}$.

It follows from the assumption that $\mathcal{F}$ is closed under projection over $\mathcal{OP}$, the fact that $\mathcal{B}$ is closed under projection over $\{\min, \max\}$, and the fact that $x = \min(+\infty, \max(x, -\infty))$ that $\mathcal{B} \circ \mathcal{F}$ is closed under projection over $\mathcal{OP} \cup \{\pm\infty\}$.

$\mathcal{B} \circ \mathcal{F}$ is the minimum set that has the above closure properties since $\mathcal{B}$ is minimumly closed over $\{\min, \max\}$, and $\mathcal{F}$ is minimumly closed over $\mathcal{OP}$. □

Corollary 4.3 *If $\mathcal{F}$ is monotonic, then $\{\mathcal{B}_\mathcal{D} \circ \mathcal{F}, \circ\}$ is a semi-group.*

The Reduction Lemma is useful in the sense that the unary function set construction, closure properties proof, and minimality proof of a given operator set which contains min and max can be reduced to the same problem over a smaller operator set. On the other hand, the known results can be expanded to a larger operator set.

The following is an application of the Reduction Lemma, other applications can be found in Section 5, and in Section 7.

Corollary 4.4 *The unary function class $\mathcal{B}_{\mathbb{R}} \circ \mathcal{L}$ is is minimumly closed over $\{\min, \max, +, \times\}$, where $\mathcal{L} = \{ax + b \mid a, b \in \mathcal{R}\}$.*

Theorem 4.5 *Trees over $\{\min, \max, +, \times\}$ of size n can be evaluated deterministically in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM.*

5 Application 1: $\oplus$-tree and Parallel Code Optimization

Definition 5.1 ($\oplus$-tree) *An $\oplus$-tree is an algebraic tree over domain integer $\mathcal{Z}$ and operator set $\{\oplus\}$ which is defined as: for all $a, b \in \mathcal{Z}$,*

$$\oplus(a,b) = \begin{cases} a+1 & a = b \\ max(a,b) & otherwise. \end{cases}$$

The $\oplus$-tree is very important in code automatic generation and optimization [21]. In fact, it is easy to convert an expression tree to an $\oplus$-tree by assigning to its left leaves the value 1 and to its right leaves the value 0, and labeling all internal nodes $\oplus$. This tree computes the *label functions* defined in [21]. An $O(\log n)$ time, n processor parallel algorithm was given in [7] where their computation rule is complicated. A simple optimal parallel algorithm is given for $\oplus$-trees.

Let $\mathcal{F} = \{f_{(a,c)} \mid a \in \mathcal{Z} \cup \{-\infty\}, c \in \mathcal{Z}\}$ be a set of unary functions, where $f_{(a,c)}$ is defined as:

$$f_{(a,c)}(x) = \begin{cases} c & x < a \\ c+1 & a \leq x \leq c+1 \\ x & x > c+1. \end{cases}$$

Claim 5.2 *$\mathcal{F}$ is minimumly closed under composition and projection over $\oplus$.*

Proof: For all $f = f_{(a,c)}$, and $g = f_{(a',c')}$: $\mathcal{F}$ is closed under composition, since

$$\begin{aligned} (g \circ f)(x) &= \begin{cases} g(c) & x < a \\ g(c+1) & a \leq x \leq c+1 \\ g(x) & x > c+1 \end{cases} \\ &= \begin{cases} f_{(a',c')}(x) & c < a' - 1 \\ f_{(a,c')}(x) & c = a' - 1 \\ f_{(-\infty,c'+1)}(x) & a' \leq c < c' + 1 \\ f_{(a,c)}(x) & c \geq c' + 1; \end{cases} \end{aligned}$$

and $\mathcal{F}$ is minimumly closed over $\oplus$ since for all $a \in \mathcal{Z}$, $f_{(a,a)} = \oplus(a, x)$ and for all $a, c \in \mathcal{D}$, $c \geq a$:

$$f_{(a,c)}(x) = (f_{(c,c)} \circ f_{(c-1,c-1)} \circ \dots \circ f_{(a+1,a+1)} \circ f_{(a,a)})(x).$$

Therefore, each function in $\mathcal{F}$ can be generated by an $\oplus$-tree. □

Theorem 5.3 (Complexity of $\oplus$-trees) *An $\oplus$-tree of size n, (the label function), can be computed in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM.*

A code over a target machine is **optimal** for an arithmetic expression, given in tree form, if the number of instructions in it is the smallest possible [21]. Combining the above theorem with the results in [7] yields the following corollary,

Corollary 5.4 *The optimal code of an arithmetic expression, given in a tree form of size n, can be constructed in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM.*

It is clear that $\mathcal{F} = \{f_{(a,c)} \mid a \in \mathcal{Z} \cup \{\pm\infty\}, c \in \mathcal{Z}\}$ is monotonic and the following lemma and theorem is a consequence of the Reduction Lemma.

Lemma 5.5 *$\mathcal{B}_{\mathcal{Z}} \circ \mathcal{F}$ is minimumly closed under composition and projection over $\{\min, \max, \oplus\}$.*

Theorem 5.6 (Complexity of a min-max-$\oplus$-tree) *Trees over $\{\mathcal{Z}, \min, \max, \oplus\}$ of size n can be computed in $O(\log n)$ time, with $n/\log n$ processors on an EREW PRAM.*

Remark 5.7 *For all $p, q \in \mathcal{Z}$, let the* step function *defined on p, q and denoted by $step_{p,q}$ be*

$$step_{p,q}(x) = \begin{cases} q & x < p \\ q+1 & x \geq p. \end{cases}$$

Then the unary function class $\mathcal{B}_{\mathcal{Z}} \circ \mathcal{F}$ used for evaluating min-max-$\oplus$-trees can be characterized as the minimum class of unary functions generated by the identity function and all step functions, which is closed under composition, min, and max.

6 Prudent Evaluation and Dynamic Processor Scheduling

An important issue in parallel computation is to develop parallel algorithms that make full use of the hardware in the parallel system. This involves the partition problem as well as the scheduling problem [8]. These problems have been proven to be *NP-hard* for the general case. Many heuristic methods have been proposed for variety problems and for some specially structured systems. In this section, it will be proven that if the unary function class for an algebraic tree is *linearly representable*, then the processor count can be reduced to $max(P(n), n)$ via load balancing.

Definition 6.1 (Linear Representability) *A unary function class $\mathcal{F}$ is (T,P)-linearly representable if $\mathcal{F}$ is closed over $\mathcal{OP}$, and for each function f in $\mathcal{F}$, size can can assigned to it, denoted by $\mathcal{SIZE}(f)$, such that:*

1. *For all $f \in \mathcal{F}$, f can be evaluated in $O(T(\mathcal{SIZE}(f)))$ time, using $P(\mathcal{SIZE}(f))$ processors.*
2. *For all $f, g \in \mathcal{F}$, $\mathcal{SIZE}(g \circ f) \leq \mathcal{SIZE}(f) + \mathcal{SIZE}(g)$. Moreover, $f \circ g$ can be computed in $T(\mathcal{SIZE}(f) + \mathcal{SIZE}(g))$ time, using $P(\mathcal{SIZE}(f) + \mathcal{SIZE}(g))$ processors.*

Theorem 6.2 *If the unary function class of a computation tree $\mathcal{T}$ is (T,P)-linear representable, then $\mathcal{T}$ can be evaluated in $O(T(n)\log n)$ time, using $\max(P(n), n)$ processors on an EREW PRAM.*

Proof: Without loss of the generality, the size of the unary function on each edge can be assumed to be a constant[1]. Initially, each node is assigned processors equal to the size of the function on the edge leaving from the node to its parent. The following two cases are presented:

[1]In order to have a reduction of $\log n$ on the processor count, the BOUNDED REDUCTION technique [10] is used. The BOUNDED REDUCTION procedure reduces the size of a tree by a factor of $\log n$. However, the size of unary functions associated with the edges in the tree might increase and not be constant. Fortunately, the linear representability ensures that the summation of the sizes of all the unary functions in the tree will not increase. Hence, the parallel *prefix sums* algorithm [2, 5, 4] can be used to assign the processors to nodes according to the size of the unary function associated with the edge leaving from the node to its parent. The rest of the argument is the same as in the following proof.

1. If P is a constant bounded function, then the proof of the theorem is trivial.

2. If P is not constant bounded, then, initially, there are n nodes and each node is assigned processors equal to the size of the unary function on the edge from the node to its parent. For the projection operation on v, the processors assigned to v are used to compute the projection, and its leaf children's processors are assigned to its non-leaf child. We use PRUDENT-COMPRESS to support the local compress. Since the rank of each node in a chain with length m can be computed in $O(\log m)$ time with m processors [2, 10], the rank of nodes in a chain can be computed by using one processor assigned to each node in the chain. The advantage of PRUDENT-COMPRESS over ISOLATED-COMPRESS is that no useless chain is produced after each application of *compress*. Any two consecutive nodes u, v are combined by using the processors associated with those two nodes and then assigning all the processors of u, v to the resulting node. Following from the instruction of this scheduling method and the linear representability of the edge function, it is easy to prove by induction that at any time, the number of processors associated with any node v is not less than the size of the function associated with the edge $(v, parent(v))$. This completes the proof. □

7 Application 2: Min-max-plus-times-division-tree

Min-max-plus-times-division tree plays an important role in algebraic computation. The min-max-plus-times-division tree contains all of the general algebraic operators. *Minus* is removed since it can be easily implemented by times and plus without increasing the magnitude of tree size. The introduction of min, max makes the arithmetic tree more general; but on the other hand, it makes the computation more complicated.

7.1 Computation Trees over $\{\mathbb{R}^+, \min, \max, +, \times, \div\}$

In this subsection, the domain of min-max-plus-times-division-tree is restricted to $\mathcal{R}^+$, the set of positive real numbers. This restriction makes the computation surprisingly easier than in the general case.

Lemma 7.1 *$\mathcal{B}_{\mathbb{R}^+\cup\{0\}} \circ \mathcal{PLF}$ is minimumly closed under composition and projection over*

$$\{\mathbb{R}^+, \min, \max, +, \times, \div\},$$

where $\mathcal{PLF} = \{\frac{ax+b}{cx+d} \mid a, b, c, d \in \mathbb{R}^+ \cup \{0\}, cd \neq 0\}$ is the set of linear fractions of nonnegative coefficients.

Proof: Note that $\mathcal{PLF}$, the set of linear fractions of nonnegative coefficient, is minimumly closed under composition and projection over $\{+, \times, \div\}$. Follows from the Reduction Lemma, it suffices to prove that $\mathcal{PLF}$ is monotonic over $\mathbb{R}^+ \cup \{0\}$. This is so because for all $f(x) = \frac{ax+b}{cx+d}$ in $\mathcal{PLF}$, the derivative of f is always positive or always negative, and the only pole of f, $-\frac{d}{c}$, is not in $\mathbb{R}^+$. □

Theorem 7.2 *Trees over $\{\mathbb{R}^+, \min, \max, +, \times, \div\}$ of size n can be computed in $O(\log n)$ time using $n/\log n$ processors, deterministically, on an EREW PRAM.*

7.2 Computation Trees over $\{\mathbb{R}, \min, \max, +, -, \times, \div\}$

The domain of a min-max-plus-times-division tree is extended from $\mathcal{R}^+$ to $\mathcal{R}$. However, since linear fractions are not monotonic over $\mathcal{R}$, the set of real numbers, the Reduction Lemma cannot be applied directly. This section will start with a short discussion of linear fractions.

Let $Lf(x) = (ax+b)/(cx+d)$ be a linear fraction. It is not hard to see that Lf has a simple pole at $x = -d/c$. In the case that $c = 0$, it is said that Lf has a pole at infinity. $LF(x)$ is one-one, iff $ad - bc \neq 0$. In this case, the derivative is either always positive or always negative. A partial function f is called **monotone increasing with respect to its pole** p (*m.i.w.t.* p, for short), if for all points x and y for which f is defined if $f(x) \leq f(y)$ then either $x \leq y < p$, $p < x \leq y$, or $y < p \leq x$. Similarly, **monotone decreasing with respect to a pole** (*m.d.w.t.*, for short) is defined. This class of functions is called the class of *pole monotonic functions*. In general, a pole is simply a point where the function is undefined, possibly ∞. Note that a linear fraction is monotone with respect to its pole. It is next stated that the monotone property is preserved under composition.

Definition 7.3 (Neighbor Interval) *For all $q \in \mathbb{R}$, $\preceq_q$, a binary relation over $\mathbb{R}$ is defined as: $a \preceq_q b$ iff either $a \leq b < q$, or $q < a \leq b$, or $a > q$ and $b < q$.*

- *If g is a monotonically increasing function with respect to a pole q, then for all $p \in \mathbb{R}$, the* **Neighbor Interval** *of g over p (denoted by $\mathcal{NI}_g(p)$) is defined as:*

  $$\mathcal{NI}_g(p) = \{x \mid \forall y \preceq_q x, g(y)\ is\ defined \Rightarrow g(y) \leq p\ \&\ \forall z \succeq_q x, g(z)\ is\ defined \Rightarrow g(z) \geq p\}$$

- *If g is a monotonically decreasing function with respect to pole q, then for all $p \in \mathbb{R}$, the* **Neighbor Interval** *of g over p (denoted by $\mathcal{NI}_g(p)$) is defined as:*

  $$\mathcal{NI}_g(p) = \{x \mid \forall y \preceq_q x, g(y)\ is\ defined \Rightarrow g(y) \geq p\ \&\ \forall z \succeq_q x, g(z)\ is\ defined \Rightarrow g(z) \leq p\}$$

Intuitively, $\mathcal{NI}_g(p)$ denotes the region on the x-coordinate such that the curve of g crosses the line $y = p$. Note that the definition of Neighbor Interval is a natural extension of the definition of the inverse of a function.

Lemma 7.4 (Composition Lemma) *If f and g are monotone functions with respect to poles p and q, respectively, then $f \circ g$ is a monotone function with respect to r, where r is any point such that $r \in \mathcal{NI}_g(p)$.*

Proof: See Section 7.4. □

The following class of unary functions are proven large enough to evaluate the min-max-plus-times-division trees. It is assumed that a tree, requiring division by zero to evaluate a subtree, need not be evaluated, and will be returned *undefined*. Thus, functions from the unary function class defined below contains undefined intervals, called **U-intervals**. Points where the function is continuous, but not necessarily differentiable, called **breakpoints**, will be maintained. These two types of closed intervals are denoted by I_i. Let $x < I_i$ denote the fact that x is less than all elements in I_i.

Definition 7.5 (Unary Function Class) *Let $\mathcal{F}$ denote the class of functions of the form*

$$f_{(Lf,I_1,\ldots,I_n,F_0,\ldots,F_{n+1},p)} = \begin{cases} F_0(x) & x < I_1 \\ F_{n+1}(x) & x > I_n \\ F_i(x) & I_i < x < I_{i+1} \text{ and } 1 \leq i < n \end{cases}$$

which satisfy the following conditions:

1. *The I_i forms a disjointed ordered set of closed intervals of two types: intervals where f is undefined, called U-intervals, and single-point intervals which are breakpoints of f.*

2. *Each F_i either agrees with Lf on its interval or is a constant function.*

3. *The function f is monotone with respect to its pole p.*

Note that functions in $\mathcal{F}$ consist of a sequence of line segments or curve segments of a linear fraction. Between each of these segments is a U-interval or a breakpoint. The size of a function in $\mathcal{F}$ is the number of U-intervals and breakpoints in it. An example of such a function is given in Figure 2.

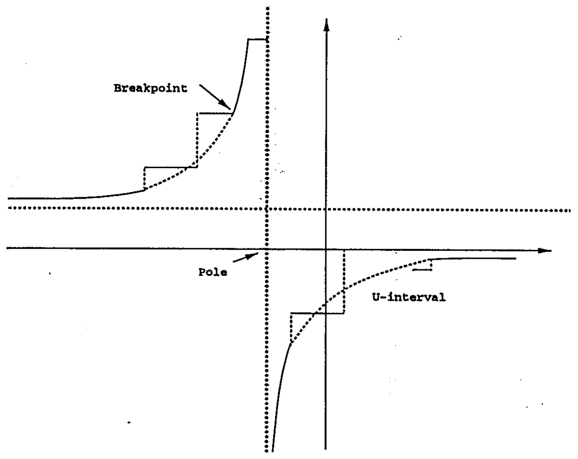

Figure 2: An Example of a Possible Unary Function for a min-max-plus-times-division tree.

7.3 Closure Properties and the Linear Representability

Lemma 7.6 *$\mathcal{F}$ is closed under composition and projection over $\{\min, \max, +, \times, \div\}$, and $\mathcal{F}$ is $(\log\log n, n)$-linear representable.*

Proof: Let $f = f_{(Lf,I_1,\ldots,I_n,F_0,\ldots,F_{n+1},p)}$ and $g = g_{(Lf,I'_1,\ldots,I'_m,F'_0,\ldots,F'_{m+1},p')}$ be two functions in $\mathcal{F}$. $\mathcal{F}$ is shown to be closed under composition with some pole, i.e., $f \circ g \in \mathcal{F}$.

For the sake of succinctness and ease of understanding, the following informal argument is given. Since the functions f and g (where defined) are a sequence of horizontal line segments or segments of a fixed linear fraction, the composition $h = f \circ g$ must be, where defined, a collection of horizontal line segments and segments of the fixed linear fraction $Lf \circ Lf'$. To see this, observe that the composition of a linear function and constant is a constant, independent of order, while the composition of two linear fractions is a linear fraction. The U-intervals and breakpoints must still be described and composition must be shown to be monotone with respect to its pole.

The first step is to compute the intervals for h. Clearly, the intervals of g will be intervals of h. But, the x in $\{x|g(x) \in I_i, 1 \leq i < n\}$ will also be in the interval of h. Note that if the preimage $g^{-1}(p)$ of a breakpoint p of f by g is an interval, then the breakpoint p will be contained in either a U-interval or a breakpoint of g. Thus, breakpoints of h are singletons but not proper intervals. We have shown that h satisfies the first two condition of the class.

By the last lemma, h is monotone with respect to some point q. Note that, in general, this point may not be the pole of $Lf \circ Lf'$; e.g., when the pole of h is not a "real" pole but just a point where h is undefined. Thus, the class of functions is closed under composition.

To see that the class is closed under projection, it is observed that following functions are in the class where a is any element $\in \mathcal{R}$: $\min(x, a)$, $\max(x, a)$, $x + a$, $x \cdot a$, and a/x. □

The number of U-intervals of h, the size of h, are counted in terms of the number of intervals of f and g. As observed above, each interval of g contributes at most one interval to h. The contribution from the intervals of f must be determined. Let I be some interval of f. It follows that I will generate a new interval for h only when some defined interval J of g is mapped onto I by g. Since g is basically a one-to-one function, there can be at most one such interval J. Using the fact that g is monotone on J, it is seen that I generates at most one U-interval for h. Therefore, the size of $g \circ f$ is no more than the summation of the size of g and f. Note that the only projection to generate U-intervals is division. Since only one breakpoint is needed between every pair of U-intervals, only two times the number of division intervals need be maintained for each part of the subtree on which we are working. In this representation it is noted that the function can be evaluated in $O(\log n)$ time.

Lemma 7.7 *For all $f \in \mathcal{F}$ of size n, and for all $x \in \mathcal{R}$, $f(x)$ can be computed in $O(1)$ time, using n processors.*

Proof: Using binary search, the interval containing x can be determined in $O(\log n)$ time; then $f(x)$ can be evaluated in constant time. However, with n processor, we can find the interval containing x in constant time. □

The following lemma is used to determine the composition of two such functions.

Lemma 7.8 *For all f and $g \in \mathcal{F}$ with size m and n respectively, $g \circ f$ can be computed in $O(\log\log(m + n))$ time, using $m + n$ processors.*

Proof: Initially, one processor is assigned to each interval of g and each interval of f. By reordering the intervals of g via a cyclic shift starting at the pole of g, their images under g will be ordered intervals which can be determined in $O(\log\log(m + n))$ time by merging finite number of sorted lists [23]. Next, the intervals of f can be merged simply with the image intervals of g. Again, this can be done in $O(\log\log(n + m))$ time using $n + m$ processors using Valliant's merging algorithm. Using this information the intervals of h can be determined. The next step is to decide whether each defined interval of h is a constant segment or a linear fraction segment. This can be done in $O(1)$ time, using $n + m$ processors. □

Corollary 7.9 *For all $c \in \mathcal{R}, f, g \in \mathcal{F}$, $\mathcal{SIZE}(f) = m$, $\mathcal{SIZE}(g) = n$, $\odot \in \{\min, \max, +, \times, \div\}$:*

1. *Let $h_\odot(x) = f(\odot(c, x))$; then $\mathcal{SIZE}(h_\odot) \leq m + 1$.*
2. *Let $h_\circ = g \circ f$; then $\mathcal{SIZE}(h_\circ) \leq m + n$.*

Theorem 7.10 *Trees over $\{\mathrm{I\!R}, \min, \max, +,, \times, \div\}$ of size n can be computed in $O(\log n \log\log n)$ time using n processors deterministically on an EREW PRAM.*

Actually, for a simple min-max-plus-times-division tree, there is a tighter bound on the maximum size of the unary function during the computation time. Observe that the projection of any non-division node does not generate a U-interval, while a division node generates only one U-interval. Therefore, the maximum size of the functions used during the evaluation of the min-max-plus-times-division tree is bounded by two times the number of division nodes. Hence, we have the following theorem.

Theorem 7.11 *Trees over $\{\mathrm{I\!R}, \min, \max, +, -, \times, \div\}$ of n nodes and d division-nodes can be computed in $O((\log n)(\log\log d))$ time using $(n \log d)/\log n$ processors.*

Note that $\mathcal{F}_\oplus = \{f_{(a,c)} \mid a, c \in \mathrm{I\!R}\}$, where $f_{(a,c)}$ is defined as:

$$f_{(a,c)}(x) = \begin{cases} c & x < a \\ c+1 & a \leq x \leq c+1 \\ x & x > c+1, \end{cases}$$

is closed under $\{\oplus\}$ defined in Section 5, and $\mathcal{F}_\oplus \subset \mathcal{F}$. Hence, $\mathcal{F}$ is also closed under composition and projection over $\{\mathrm{I\!R}, \min, \max, +, -, \times, \div, \oplus\}$. Thus:

Theorem 7.12 *Trees over $\{\mathrm{I\!R}, \min, \max, +, -, \times, \div, \oplus\}$ of n nodes can be computed in $O(\log n \log\log n)$ time using n processors on an EREW PRAM.*

7.4 The Proof of Lemma 7.2

We first present some definitions.

Definition 7.13 (α-Transformation) *An α transformation at p, denoted by α_p is a Mobius transformation of the form $\alpha_p(x) = \frac{px}{x-p}$.*

Definition 7.14 (β-Transformation) *A β transformation at p, denoted by β_p is a Mobius transformation of the form $\beta_p(x) = \frac{px}{x+p}$.*

Lemma 7.15 *For all $p \in \mathrm{I\!R}$ and for all $x \in \mathrm{I\!R}$: (i) $\alpha_p(\alpha_p(x)) = x$, (ii) $\alpha_p(\beta_p(x)) = -x$, (iii) $\beta_p(-\beta_p(x)) = -x$, and (iv) $\beta_p(-\alpha_p(x)) = x$.*

Lemma 7.16 *For all $p \in \mathrm{I\!R}$ and for all $x \in \mathrm{I\!R}$: (i) $\alpha_p(x) \leq p$ iff $x \leq p$, and (ii) $\beta_p(x) \leq p$ iff $x \geq -p$.*

Theorem 7.17 **(α-β-Transformation Theorem)** *For all unary functions $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and for all $p \in \mathbb{R}$:*

1. *f is monotone with respect to its pole p (m.r.t. p, for short) iff $f \circ \alpha$ is monotonic.*
2. *f is monotone with respect to its pole p iff $f \circ \beta$ is monotonic.*

Proof: For all $x_1, x_2 \in \mathbb{R}$:

$$\alpha_p(x_2) - \alpha_p(x_1) = \frac{p^2(x_1 - x_2)}{(x_1 - p)(x_2 - p)}.$$

It follows that,

$$\begin{array}{ll}
& f \textit{ is monotone } (\diamond) \textit{ with respect to } p. \\
\Leftrightarrow & \left\{ \begin{array}{lll} x_2 \le x_1 < p & \Rightarrow & f(x_2) \diamond f(x_1) \\ x_2 < p < x_1 & \Rightarrow & f(x_1) \diamond f(x_2) \\ p < x_2 \le x_1 & \Rightarrow & f(x_2) \diamond f(x_1) \end{array} \right. \qquad (\textit{by definition}) \\
\Leftrightarrow & \left\{ \begin{array}{lllll} x_2 \le x_1 < p & \Leftrightarrow & \alpha(x_1) \le \alpha(x_2) < p & \Rightarrow & f(\alpha(x_1) \diamond f(\alpha(x_1)) \\ x_2 < p < x_1 < p & \Leftrightarrow & \alpha(x_2) < p < \alpha(x_1) & \Rightarrow & f(\alpha(x_1) \diamond f(\alpha(x_1)) \ (\textit{by Lemma } 7.16) \\ p < x_2 \le x_1 & \Leftrightarrow & p < \alpha(x_1) < \alpha(x_2) & \Rightarrow & f(\alpha(x_1) \diamond f(\alpha(x_1))J \end{array} \right. \\
\Leftrightarrow & f \circ \alpha_p \textit{ is monotonic.}
\end{array}$$

where $\diamond$ ='$<$' if f is monotonicly-increasing with respect to p; and $\diamond_g$ ='$>$' if f is monotonicly-decreasing with respect to p. Similarly, it can be proven that f is monotone with respect to its pole p iff $f \circ \beta$ is monotonic. □

In fact, a stronger result is stated as: for all $q \in \mathbb{R}$, f is monotone with respect to its pole p iff $f \circ \frac{px}{x+q}$ is monotonic.

The α-β-Transformation Theorem (Theorem 7.17) provides an alternative way of defining the pole monotonic function.

Definition 7.18 *A function f is a pole monotonic function with respect to pole p if $f \circ \alpha_p$, or $f \circ \beta_p$ is a monotonic function.*

Theorem 7.19 **(Composition Lemma)** *If f and g are monotone with respect to poles p and q, respectively, then $f \circ g$ is monotone with respect to r, where r is any point such that $r \in \mathcal{NI}_g(p)$.*

Proof: Since if g is a pole monotonic function of pole q, then for all $p \in \mathbb{R}$,

$$g'(x) = \left\{ \begin{array}{ll} p & x \in \mathcal{NI}_g(p) \\ g(x) & otherwise \end{array} \right.$$

is also a pole monotonic function with pole q. Therefore, we need consider only the case where for all $r \in \mathcal{NI}_g(p)$, $g(r) = p$. First, the lemma is proven true for the case when g is monotonic, i.e. the pole of g is ∞.

Lemma 7.20 (**∞-pole-g Lemma**) *If f is a pole monotonic function of pole p and g is a monotonic function, then for all $r \in \mathcal{NI}_g(p)$, $f \circ g$ is a pole monotonic function with pole r.*

Proof: Assume that g is $\Diamond_g$-monotonic, where $\Diamond_g =$'$<$' for monotonicly-increasing and $\Diamond_g =$'$>$' for monotonicly-decreasing. Assume also that f is $\Diamond_f$ pole monotonic. The lemma follows from the analysis of the three cases below,

- If $x_2 \leq x_1 \leq r$, then $g(x_2)\Diamond_g g(x_1)\Diamond_g p$. Therefore, $(f \circ g)(x_2)\underline{\Diamond_f \oplus \Diamond_g}(f \circ g)(x_1)$.
- If $x_2 \leq r \leq x_1$, then $g(x_2)\Diamond_g p \Diamond_g g(x_1)$. Therefore, $(f \circ g)(x_1)\underline{\Diamond_f \oplus \Diamond_g}(f \circ g)(x_2)$.
- If $r \leq x_2 \leq x_1$, then $p\Diamond_g g(x_2)\Diamond_g g(x_1)$. Therefore, $(f \circ g)(x_2)\underline{\Diamond_f \oplus \Diamond_g}(f \circ g)(x_1)$.

where $\oplus$ is a binary operator defined as: (1) '$<$'$\oplus$'$<$'='$<$', (2) '$<$'$\oplus$'$>$'='$>$', (3) '$>$'$\oplus$'$<$'='$>$', and (4) '$>$'$\oplus$'$>$'='$<$'. □

Because of the α-β-Transformation Theorem (Theorem 7.17), it suffices to show that $f \circ g \circ \beta_r$ is monotonic, when g is a general pole monotonic function.

Since g is a monotonic function with pole q, $g \circ \alpha_q$ is a monotonic function. Therefore, by Lemma 7.20 $f \circ g \circ \alpha_q$ is a pole monotonic function with pole $r' \in \mathcal{NI}_{g \circ \alpha_q}(p)$; i.e. $g(\alpha_q(r')) = p$. Hence:

1. $f \circ g \circ \alpha_q \circ \alpha_{r'}$ is monotonic.
2. there exists $r' \in \mathcal{NI}_{g \circ \alpha_q}(p)$ such that $\alpha_q(r') = r$.

Note that $(\alpha_q \circ \alpha_{r'})(x) = \frac{qr'x}{(r'-q)x+qr'} = \beta_r(x)$. Hence, by (2) we have, $f \circ g \circ \beta_r$ is monotonic. Therefore, $f \circ g$ is a pole monotonic function with pole r. □

8 Conclusion and Open Problems

In this paper, a systematic method for tree-based parallel algorithm generation was presented, and some nontrivial sufficient conditions for the existence of efficient parallel algorithms for tree computation were proven. This provides a powerful tool for designing parallel algorithms for many problems.

There are two interesting open questions;

1. Can the general min-max-plus-times-division tree be evaluated in $O(\log n)$ time using $n/\log n$ processors?
2. What is the condition necessary for evaluating a computation tree in polylogarithmic time, using only a polynomial number of processors?

Acknowledgments

We would like to thank Hillel Gazit, John Reif, and Margaret Reid-Miller for invaluable help and discussions.

References

[1] K. Abahamson and N. Dadoun and D. K. Kirkpatrick and T. Przytycka. A Simple Parallel Tree Contraction Algorithm. *Journal of Algorithms*, 10(2): 287–302, 1989.

[2] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, January 1990.

[3] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal Assoc. Computing Machinery*, 21(2):201–208, April 1974.

[4] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *27th Annual Symposium on Foundations of Computer Science*, pages 478–491, IEEE, Toronto, Oct 1986.

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

[6] S. A. Cook. Towards a complexity theory of synchronous parallel computation. In *Internationales Symposium uber Logik und Algorithmik zu Enren von Professor Hort Specker*, page , February 1980.

[7] E. Dekel, S. Ntafos, and S.-T. Peng. *Parallel Tree Techiques and Code Opimization*, pages 205–216. Volume 227 of *Lecture Notes in Computer Science*, Springer-Verlag, 1986.

[8] D. D. Gajski and J. K. Peir. Essentail issues in multiprocessors systems. *IEEE Computer.*, 9–27, 1985.

[9] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *27th Annual Symposium on Foundations of Computer Science*, pages 492–501, IEEE, Toronto, October 1986.

[10] H. Gazit, G. L. Miller, and S.-H. Teng. Optimal tree contraction in an EREW model. *Proceedings 1987 Princeton Workshop on Algorithm, Architecture and Technology: Issues for Models of Concurrent Computation*, pp 139–156, 1987.

[11] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic eveluation and its applications. in *Proceedings 6th Conference on Foundations of Software Technology and Theoretical Computer Science.* Lecture Notes in Computer Science 241, pp453-469, 1986.

[12] P. B. Gibbons and R. M. Karp and G. L. Miller and D. Soroker. Subtree isomorphism is in random NC. *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88.* Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, N.Y., ed. J. H. Reif, pp43–52, 1988.

[13] X. He and Y. Yesha. Binary tree algebraic comptations and parallel algorithms for simple graphs. *J. Algorithms*, 9(1), 92–113, 1988.

[14] A. Karlin and E. Upfal. Parallel hashing–an efficient implementation of shared memory. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 160–168, ACM, Berkeley, May 1986.

[15] P. Klein and J. Reif. An efficient parallel algorithm for planarity. In *27th Annual Symposium on Foundation of Computer Science*, pages 465–477, IEEE, Oct 1986.

[16] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computation by ranking (Extended Abstract), in *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88,* Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, N.Y., ed. J. H. Reif, pp 101–110, 1988.

[17] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, IEEE, Portland, Oregon, 1985.

[18] G. L. Miller and S.-H. Teng. Dynamic parallel complexity of computational circuits. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 254–264, ACM, New York, May 1987.

[19] G. L. Miller and S.-H. Teng. Systematic methods for tree based parallel algorithm development. In *Second International Conference on Supercomputing*, pages 392–403, Santa Clara, May 1987.

[20] A. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194, IEEE, Los Angeles, Oct 1987.

[21] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *JACM*, 17(4):715–728, Oct. 1970.

[22] S.-H. Teng and B. Wang. Parallel algorithms for message decomposition. *Journal of Parallel and Distributed Computing*, 4():231–249, June 1987.

[23] L. G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, ACM, 1981.

[24] U. Vishkin. *Synchronous Parallel Computation – a Survey.* TR 71, NYU, 1983.

A Two Procedures for LOCAL-COMPRESS

For each vertex v with children $v_1, \ldots, v_k$, k locations, $l_1, \ldots, l_k$, are set aside in the common memory. Initially, each l_i is empty or *unmarked.* When the value of v_i is known, the value will be assigned to l_i; The assignment is denoted by *mark* l_i in the following procedure. Let $Arg(v)$ denote the number of unmarked l_i. Then, initially, $Arg(v) = k$, the number of children of v. Let *vertex* $(P(v))$ be the vertex associated with the sole parent of v with storage location $P(v)$. All vertices shalled be *tagged* with one of four possibilities: G, M, R, or $\emptyset$. Vertices with a nonempty tag belong to an isolated chain. When a chain is first isolated, the root is tagged R, the tail is tagged G, and the vertices between the root and the tail are tagged M. A vertex v is *free* if $Arg(v) = 1$ and $Tag(v) = \emptyset$; otherwise, v is *not free.*

```
Procedure ISOLATE-COMPRESS [10]
    in parallel for all $v \in V - \{r\}$ do
        case $Arg(v)$ equals
        0)  Mark $P(v)$ and delete $v$ ;;
        1)  case $Tag$(v) equals
            $\emptyset$)  if child is not free and parent is free then $Tag(v) \leftarrow G$
                if parent is not free and child is free then $Tag(v) \leftarrow R$
                if parent is free and child is free then $Tag(v) \leftarrow M$ ;;
            $G$)  if $Tag(P(v)) = R$ then $Tag(v) \leftarrow \emptyset$
                $f_{v,P(P(v))} = f_{P(v),P(P(v))} \circ f_{v,P(v)}$; $P(v) \leftarrow P(P(v))$ ;;
            $M$)  if $Tag(P(v)) = R$ then $Tag(v) \leftarrow R$ ;;
                $f_{v,P(P(v))} = f_{P(v),P(P(v))} \circ f_{v,P(v)}$; $P(v) \leftarrow P(P(v))$
            esac
        esac
```

```
Procedure PRUDENT-COMPRESS [10]
    1) form isolated chains from free vertices
    2) in parallel for all isolated chains $C$ in 1) do
        COLLAPSE($C$)
```

*COLLAPSE(C)*is a procedure with computed the ranks of all nodes in an isolated chain C and uses this information to compress the chain in such way that no useless subchain is produced. *COLLAPSE(C)* can be specified as:

```
Procedure COLLAPSE($C$)
    1) run list-ranking on $C$.
    2) while $C$ is not a singleton do
        compress $C$ by combining the vertex of rank $2i - 1$
        with the vertex of rank $2i$
        to form a new vertex of rank $i$.
    3) set it free.
```

Lemma A.1 [2] *By using the ISOLATE-COMPRESS procedure or the PRUDENT-COMPRESS procedure, a tree contraction algorithm can be implemented on an EREW PRAM without increasing the time count and processor count.*

Lemma A.2 [3] *The PRUDENT-COMPRESS procedure is at most two times slower than the ISOLATE-COMPRESS procedure. However, no useless chain is produced during the application of a tree contraction algorithm. It is called* prudent compress.

[2]See the LOCAL COMPRESS Lemma in [10].
[3]See Lemma 4.1 in [10].