From: CONCURRENT COMPUTATIONS
Edited by Stuart K. Tewsburg, Bradley W. Dickinson, and Stuart C. Schwartz
(Plenum Publishing Corporation, 1988)

Chapter 9

Optimal Tree Contraction in the EREW Model [1]

Hillel Gazit [2]
Gary L. Miller [2]
Shang-Hua Teng [2]

Abstract

A deterministic parallel algorithm for parallel tree contraction is presented in this paper. The algorithm takes $T = O(n/P)$ time and uses P ($P \leq n/\log n$) processors, where n = the number of vertices in a tree using an Exclusive Read and Exclusive Write (EREW) Parallel Random Access Machine (PRAM). This algorithm improves the results of Miller and Reif [MR85,MR87], who use the CRCW randomized PRAM model to get the same complexity and processor count. The algorithm is *optimal* in the sense that the product $P \cdot T$ is equal to the input size and gives an $O(\log n)$ time algorithm when $P = n/\log n$. Since the algorithm requires $O(n)$ space, which is the input size, it is *optimal* in space as well. Techniques for prudent parallel tree contraction are also discussed, as well as implementation techniques for fixed-connection machines.

[1] This work was supported in part by National Science Foundation grant DCR-8514961.
[2] University of Southern California, Los Angeles, CA

9.1 Introduction

In this paper we exhibit an optimal deterministic Exclusive Read and Exclusive Write (EREW) Parallel Random Access Machine (PRAM) algorithm for parallel tree contraction for trees using $O(n/P)$ time and P ($\mathbf{P} \leq n/\log n$) processors. For example, we can dynamically evaluate an arithmetic expression of size n over the operations of addition and multiplication using the above time and processor bounds. In particular, suppose that the arithmetic expression is given as a tree of pointers where each vertex is either a leaf with a particular value or an internal vertex whose value is either the sum or product of its children's values. These time and processor bounds also apply to arithmetic expressions given as a tree, as previously described. One can reduce an expression given as a string to one given in terms of a pointer by using the results of Bar-On and Vishkin [BV85]. There are many other applications of our parallel programming technique for problems that possess an underlying tree structure, such as the expression evaluation problem [MR]. The goal of this paper is to improve this paradigm so that the $O(\log n)$ time $n/\log n$ processor algorithm can easily and efficiently be constructed for a wide variety of problems.

Our algorithm has two stages. The first stage uses a new reduction technique called Bounded (Unbounded) Reduction. A bounded (unbounded) degree tree of size n is reduced to one of size P in $O(n/P)$ time, using P processors on an EREW PRAM. The second stage uses a technique called Isolation to contract a tree of size $\mathbf{P}$ to its root in $O(\log \mathbf{P})$ time, using $\mathbf{P}$ processors on an EREW PRAM.

The constants are small since techniques notorious for introducing large constants, such as expander graphs and the more general workload balancing techniques of Miller and Reif [MR85,MR87], are not used. Instead, only one simple load balance is needed to support our procedure, ***UNBOUNDED-REDUCTION.*** Wyllie's technique for list ranking, when used to carry out the Compress operation, performs many unnecessary function compositions [MR87]. In Section 9.6.4 we show that these techniques provide a solution for prudently compressing chains . In Section 9.3 we discuss how to implement these procedures on a fixed connection machine, which minimizes the size of constants.

Miller and Reif [MR85,MR87] give a deterministic Concurrent Read and Concurrent Write (CRCW) PRAM algorithm for tree contraction, using $O(\log n)$ time and n processors, and an 0-sided randomized version (CRCW) of the algorithm using $n/\log$n processors. By attaching a complete binary tree to each path of the original tree to guide the application of Compress, Dekel et al. [DNP86] present a tree contraction algorithm on an EREW PRAM in $O(\log n)$ time using n processors. But their methods and proofs are complicated and difficult to follow. This paper, on the other hand, presents a parallel-tree contraction algorithm for an EREW PRAM, which reduces the processor count to $n/\log$n and simplifies their algorithm.

This paper consists of *six* sections. Section 9.2 contains basic graph theoretic results and definitions that are needed in Section 9.4 to reduce the problem of size n to one of size n/P, where P is the number of processors. In Section 9.3 we discuss the relationship between the List-Ranking problem and the All-Prefix-Sums problem. In Section 9.4 we show how our reduction of a tree of size n to one of size $n/\log n$ can be performed with only one List-Ranking. Section 9.5 reviews the work of Miller and Reif [MR85] that is used in Section 9.6. Section 9.5 also includes definitions of Rake and Compress, the two basic tree contraction operations, presented by Miller and Reif [MR85,MR87]. In Section 9.6, the isolation technique is used to implement parallel tree contraction on a deterministic EREW PRAM in $O(\log n)$ time by using $n/\log n$ processors. In that section, we demonstrate that the isolation technique can be use for prudent parallel tree contraction,

9.2 Basic Graph Theoretic Results

The main graph theoretic notions needed in this paper are defined in this section. We also present a simple, yet important, structural theorem for trees which is used to reduce a tree of size n to one of size $O(n/P)$.

We begin with some basic definitions. Throughout this paper a **tree, T = (V, E),** is defined as a directed graph, in which every vertex except the root points to its unique parent. The **weight** of a vertex v in T is the number of vertices in the subtree rooted at v, denoted by $W(v)$. If n equals the number of vertices in T, the weight of the root r is n.

In this section we also consider the decomposition of a tree T into subtrees by partitioning the tree at its vertices, which partitions the edges of T in a natural way. Subtrees (subgraphs) are then formed out of each set of edges by reintroducing the end-point vertices. These subgraphs are known as **bridges.** The standard formal definition of a bridge will be presented next.

Let C be a subset of V in a graph, G = (V, E). Two edges, e and e' of G, are **C-equivalent** if there exists a path from e to e' which avoids the vertices C. The induced graphs, formed from the equivalent classes of the C-equivalent edges, are called the **bridges** of C. A bridge is **trivial** if it consists of a single edge. The **attachments** of a bridge B are the vertices of B in C.

Let **m** be any integer such that $1 < \mathbf{m} \leq$ n. One could think of $\mathbf{m} = n/P$, where P is the number of processors; but the fact that $m = n/p$ is not used in this section. A vertex is **m-critical** if it belongs to the following set of vertices of T:

$$C = \{v \in V \mid \lceil \frac{W(v)}{\mathbf{m}} \rceil \neq \lceil \frac{W(v')}{\mathbf{m}} \rceil \text{ for all children v' of v}\}.$$

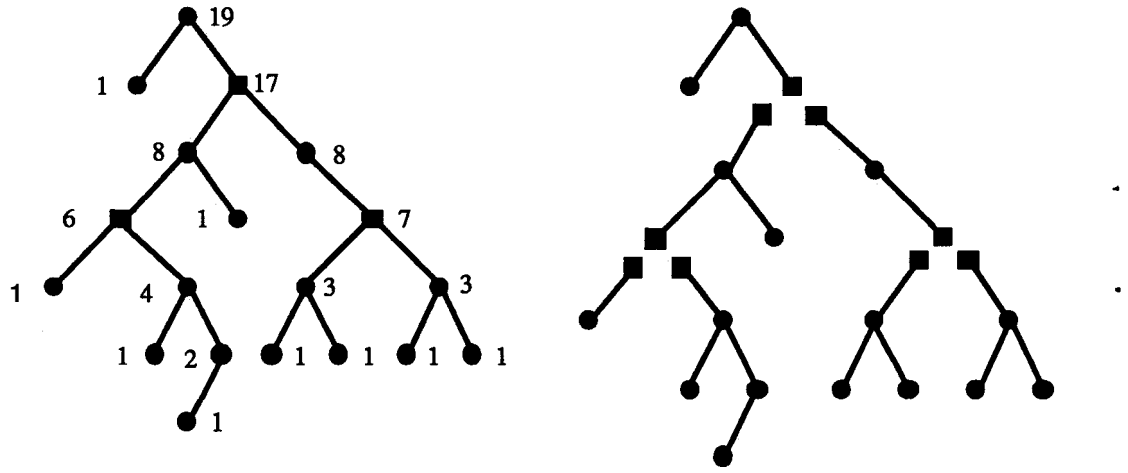

Figure **9.1:** The Decomposition of a Tree into its 5-Bridges.

The m-bridges are those bridges of C in T where C is the set of rn-critical vertices of T. Note that an attachment of an rn-bridge B is either the root of B or one of its leaves. In Figure **9.1** we give a tree and its decomposition into its 5-bridges. The vertices represented by boxes are the 5-critical vertices, and the numbers next to these vertices are their weights. Next, we show that B can have at most one attachment which is not its root.

Lemma 9.2.1 *If* B *is* an m-bridge *of* a tree T, then B can have at most one attachment, *i.e.,* a leaf *of* B.

Proof: The proof is by contradiction. We assume that B is an m-bridge of a tree T, and v and v' are two leaves of B that are also m-critical. We prove that this is impossible. Let w be the lowest common ancestor of v and v' in T. Since B is connected, w must be a vertex of B and cannot be m-critical, because if that were true, then both v and v' would not belong to B. On the other hand, w must be m-critical since w has at least two children of weight m. □

From Lemma **2.1** one can see that there are three types of m-bridges: (1) a leaf bridge which is attached by its root; (**2**) an edge bridge which is attached by its root and one leaf; and (3) a top bridge, containing the root of T, which exists only when the root is not m-critical. Except for the top bridge, the root of each m-bridge has **a** unique child. The edge from this child to its root is called the leading edge of the bridge.

Lemma 9.2.2 The number *of* vertices *of* an m-bridge is at most m + **1.**

Proof: Consider the three types of m-bridges: leaf, edge, and top. If B is a leaf bridge then its root r' is the first and only vertex in B with weight $\geq$ m. Since

$rn > 1$, there must exist a unique vertex w in B which is the only child of r'. Thus, B consists of the subtree rooted at w plus r'. Since the weight of w is less than m, the number of vertices of B is at most rn. If B is a leaf rn-bridge with rn-critical root r' and rn-critical leaf u, then r' will have a unique child w in R. Therefore, the number of vertices in B is $W(w) - W(u) + 2$. $W(w) - W(u) < rn$ since w is not rn-critical. Thus, the number of vertices in B $\leq rn + 1$. The case for a top bridge follows by similar arguments. □

Although it is desirable to have few rn-bridges [i.e., $O(n/m)$], that is not the case here. This fact can be seen in the example of an unbounded degree tree of height 1 where $rn < n$ and every edge is an m-bridge. However, the number of rn-critical vertices is not large.

Lemma 9.2.3 *The number of rn-critical nodes in a tree of size n is at most* $2n/m - 1$ *for* $n \geq rn$.

Proof: Let n_k be the number of nodes in a minimum size tree with k rn-critical nodes. The lemma is equivalent to the statement:

$$n_k \geq \left(\frac{k+1}{2}\right)rn \quad \text{for} \quad k \geq 1. \tag{9.2.1}$$

Inequality **9.2.1** is proven by induction on k. If v is rn-critical, then its weight must be at least rn. This proves **9.2.1** for $k = 1$. Suppose that **9.2.1** is true for $k \geq 1$ and all smaller values of k. We prove **9.2.1** for $k + 1$. Suppose that T is a minimum size tree with $k + 1$ rn-critical nodes. The root r of T must be rn-critical for it to be of minimal size, because we can discard all of the tree above the first rn-critical node (the root bridge) without affecting the number of critical nodes. Assuming r is rn-critical, there are two possible cases for the children of root r: (1) r has two or more children, $u_1, \ldots, u_t$, and each of their subtrees contains an rn-critical node; or (2) r has exactly one child u whose subtree contains an rn-critical node.

We first consider Case 1. Let n_i be the number of vertices, and k_i the number of m-critical nodes in the subtree of u_i for $1 \leq i \leq t$. Since T is of minimum size, $u_1, \ldots, u_t$ must be the only children of r. Thus, $k + 1 = \sum_{i=1}^{t} k_i$ and $\sum_{i=1}^{t} n_i \leq n_{k+1}$. Using these two inequalities and the inductive hypothesis we get the following chain of inequalities:

$$\begin{aligned} n_{k+1} &\geq \sum_{i=1}^{t} n_i \geq \sum_{i=1}^{t} \left(\frac{k_i+1}{2}\right) m \geq \left(\frac{(\sum_{i=1}^{t} k_i)+t}{2}\right) m, \\ &\geq \left(\frac{k+t}{2}\right) m \geq \left(\frac{k+2}{2}\right) m \geq \left(\frac{(k+1)+1}{2}\right) m. \end{aligned}$$

This proves Case 1.

In Case 2 the subtree rooted at u contains a unique maximal node w which is rn-critical, and the subtree of w contains k rn-critical nodes. Thus, the induction

hypothesis shows that $W(u) \geq \lceil \frac{k+1}{2} \rceil m$. Since $W(r)$ is an integral multiple of m greater than $W(u)$,

$$W(r) \geq (\lceil \frac{k+1}{2} \rceil + 1)m \geq \lceil \frac{k+2}{2} \rceil m.$$

□

The m-contraction of a tree, T with root r is a tree, T_m = (V', E'), such that the vertices V' are the rn-critical vertices of T union r. Two vertices, v and v' in V', are connected by an edge in T_m, if there is an rn-bridge in T which contains both v and v'. Note that every edge in T_m corresponds to a unique rn-bridge in T, which is either an edge bridge or the top bridge. Thus by **Lemma** 9.2.3, **T' is a tree with** at most $2n/m$ vertices. In the next section we show how to reduce a tree to its m-contraction, where $m = \mathrm{n/P}$ in $O(m + \log n)$ time.

9.3 List-Ranking Versus All-Prefix-Sums

There are two problems which are very similar, but their complexity is quite different. The first is the List-Ranking problem, where one is given a linked list of length n packed into consecutive memory locations. The goal is to compute for each pointer its distance from the beginning of the list. The second problem is **All-Prefix-Sums,** where we are given a semi-group (S,$*$) and a string of elements, $s_1 \ldots s_n$ We may request that the n elements be loaded into memory in some convenient order. The solution is to replace each element s_i with $t_i = s_1 * \cdots * s_i$. It is easy to see how to generalize the List-Ranking problem to include the All-Prefix-Sums problem by storing s_i in the i^{th} pointer and requiring all prefixes to be computed. All known algorithms for List-Ranking solve this generalized problem for semi-groups at no extra cost. Thus, we could view any All-Prefix-Sums problem as a List-Ranking problem; but this may increase the running time.

It can be shown that an All-Prefix-Sums problem can be computed in 6log n time on a binary N-cube parallel computer where N = $n/\log$ n. On the other hand, the results for List-Ranking use the PRAM model. The problem was first introduced by Wyllie **[Wyl79].** He gave an $O(\log \mathrm{n})$ time n processor algorithm for an EREW **PRAM.** This result has been improved upon by many authors. Miller and Reif **[MR85]** give the first $O(\log n)$ time, optimal number of processors algorithm for this problem. Their algorithm uses randomization and requires the CRCW model. Cole and Vishkin [CV86b] give the first $O(\log n)$ time deterministic EREW algorithm, using an optimal number of processors. Both of these algorithms involve very large constants. Miller and Anderson [AM87] give a simple deterministic EREW algorithm and an even simpler, randomized EREW $O(\log n)$ time, optimal algorithm for Lirt-Ranking.

While the List-Ranking problem can be performed in $O(\log n)$ time on an EREW

PRAM using $n/\log n$ processors, it translates into an $O(\log^2 n)$ algorithm on a binary N-cube, i.e., $N = n/\log n$. Therefore, if the ultimate purpose of a parallel algorithm is to run it on a fixed connection machine, then we should minimize the number of List-Rankings we perform; and, whenever possible, replace the List-Ranking procedure with the All-Prefix-Sums procedure. Karlin and Upfal [KU86] show that once the numbering is known, then the values can be loaded into consecutive memory locations in $O(\log n)$ time by using a randomized algorithm. Readers who are interested in the subject should see the improved results of Ranade [Ran87].

We shall present our code so that, whenever possible, we can perform the All-Prefix-Sums problem on strings stored in consecutive memory locations. Thus, if the user implements these sums on a fixed connection machine, he can implement them only in $O(\log n)$ time. We shall refer to All-Prefix-Sums as all prefix sums over consecutive memory locations.

Tarjan and Vishkin [TV85] define the notion of a Euler tree tour of an ordered tree. Recall that we have defined a tree as a directed graph consisting of directed edges from child to parent. Let T' be the tree T where we have added in the reverse edges. A **Euler tree tour** is the path in T' from root to root which traverses the edges around the outside of T in a clockwise fashion, when T' is drawn in the plane in an order preserving way.

Figure 9.2 shows how to compute the weights of all nodes in a tree, in parallel, using the Euler tree tour. This algorithm for computing weights has been derived from Tarjan and Vishkin [TV85].

It is an interesting open question whether parallel tree contraction can be performed with only one List-Fbnking or not. In the next section we show that the m-contraction of a tree of size n can be constructed with only one List-Ranking of the Euler tree tour.

Procedure *WEIGHTS*(T)

1. Number every tree edge 1, and its reverse 0.
2. Compute the All-Prefix-Sums of the Euler tree tour.
3. Compute the weight of each vertex v as the difference between the prefix sums when we first visited v and when we last visited w.

Figure 9.2: Computing the Weights.

9.4 Reduction From Size n to Size n/m

In this section we show how to contract a tree of size n to one of size n/m in $O(m)$ time using n/m processors, for $m \geq \log n$. If we set $m = \lceil n/P \rceil$, then this gives us a reduction of a problem of size n to one of size P. In Section **9.6** we show how to contract a tree of size P to a point. In that section we consider the special case of when tree T is of bounded degree. In Subsection **9.4.2** the general case of when the tree may be of unbounded degree is discussed.

Let us assume that a tree is given as a set of pointers from each child to its parent and that the tree is ordered, so that the children of a vertex are ordered from left to right. Further, assume that each parent has a consecutive block of memory cells, one for each child, so that each child can write its value, when known, into its location. This last assumption permits us to use the All-Prefix-Sums procedure to compute the associative functions of each set of siblings. This assumption is used to determine the rn-critical sets.

9.4.1 The Bounded Degree Case

From Section 9.2, we learned that there are at most $2n/m$, rn-critical vertices. Since we assume in this section that the tree is of bounded degree d, there can be at most d of the rn-bridges common to and below an rn-critical vertex of T. We also know that each rn-bridge has a size of at most $m + 1$. To perform the reduction, we need only find the m-bridges and efficiently assign them to processors in order to evaluate them. A processor, assigned to each m-critical vertex, computes the value (function) of the rn-bridges below it. A processor is also assigned to each existing root bridge, and computes the function for each root bridge. This algorithm is given in procedural form in Figure **9.3.**

We discuss in more detail how to implement the steps in Procedure ***BOUNDED-REDUCTION.*** We start with Step **2** in which the Euler tree tour of T is loaded into consecutive memory locations. (This step is described in Section **9.3.)** This representation is used in all steps except Step **4.** Step **3** is also described in Section **9.3.** To compute the rn-critical vertices (Step **4),** we copy the weight of each vertex back to the original representation. There, we compute the maximum value of each set of siblings by using the All-Prefix-Sums procedure on this representation in the natural way. The maximum value is then returned to the right-most sibling. Note that the maximum value could have been returned to all siblings, which would have allowed all vertices to determine if their parents were m-critical with no extra message passing. The right-most sibling could then determine whether its parent is an m-critical vertex or not. To enumerate the rn-critical vertices, we can use either representation of the tree.

Procedure *BOUNDED-REDUCTION*(T)

1. **set** $m \leftarrow \lceil n/2P \rceil$
2. Compute a List-Itanking of the Euler tree tour of T. Use these values to map the i^{th} edge into memory location i
3. Using the All-Prefix-Sums procedure, compute the weight of every vertex in T.
4. Using the All-Prefix-Sums procedure over the original representation, determine the rn-critical vertices in T.
5. Using the All-Prefix-Sums procedure, assign a processor to each rn-critical vertex and one to the root.
6. Require each processor assigned in Step 5 to compute the value of the leaf bridges below it and the unary function for the edge or top bridges below it.
7. Return the rn-contraction of T and store the Euler tree tour in consecutive memory locations.

Figure 9.3: A Procedure that Contracts a Bounded Degree Tree of Size n to One of Size n/P.

In Step 6 we note that each leaf of the rn-bridge is stored in at most $2m$ consecutive memory locations: i.e., there is one memory location for each edge or its reverse. On the other hand, the edge and root rn-bridges consist of two consecutive runs of memory locations with a total size of $2m$. If we are implementing this algorithm on a fixed connection machine, then the memory cells of each rn-bridge are contained in the memory of a constant number of processors. In Step 7 we note that the Euler tree tour of the rn-contraction of T can be constructed without using List Ranking.

9.4.2 The Unbounded Degree Case

In this subsection we show how to compute the rn-contraction of an unbounded degree tree. In the unbounded case the number of leaf rn-bridges may be much larger than the number of processors. On the other hand, the number of bridges that are either a top bridge or an edge bridge is bounded by the number of m-critical vertices, which, in turn, is bounded by $2n/m$. In the unbounded degree case, a processor may be required to evaluate many small leaf bridges, since there may be a large number of them. The procedure, *UNBOUNDED-REDUCTION,* is given in Figure 9.4.

Steps 1 4 are identical to those used in the *BOUNDED-REDUCTION* procedure. Steps 5 and 6 assign a processor to a collection of leaf bridges. Step 5 is a straightforward, All-Prefix-Sums calculation. Note that all leaf bridges in an interval, $[(i -$

Procedure *UNBOUNDED-RED UCTION(T)*

1. **set** m $\leftarrow \lceil n/2P \rceil$
2. Compute a List-Ranking of the Euler tree tour of T and map the i^{th} edge into memory location i.
3. Using the All-Prefix-Sums procedure, compute the weight of every vertex in T.
4. Using the All-Prefix-Sums procedure over the original representation, determine the m-critical vertices in T.
5. Assign to each leading edge of a leaf bridge a value equal to the weight of its bridge. To all other edges, assign a value of zero. Compute All-Prefix-Sums of value; let $S(e)$ be the sum up to e.
6. Assign processor i to all leaf bridges with leading edge e so that $(i-1)m \leq S(e) < im$.
7. Using the All-Prefix-Sums procedure, assign a new processor to each edge or root m-bridge.
8. Require each processor assigned in Step 7 to compute the value (unary function) of the leaf bridge (edge or root) for its assigned bridge.
9. Return the m-contraction of T and store the Euler tree tour in consecutive memory locations.

Figure 9.4: A Procedure that Contracts an Arbitrary Tree of Size n to One of Size n/P.

$1)m, im)$, are stored in consecutive memory locations. Therefore, we need assign a processor only to the first leaf bridge in each interval. After Step 5, the first leading edge of each interval knows that it is the first leading edge. Therefore, we can do one more All-Prefix-Sums calculation to enumerate the leading edges that are the first ones in their interval. Using this information, we can then assign the processors per Step 6. Note, in Step 8, that each processor must evaluate at most 2m vertices.

Another way of assigning processors to bridges is to compute the weight of all m-bridges and use Step 6 to assign processors to all bridges–not just leaf bridges. The weight of a leaf bridge is the difference between its bottom attachment and its top attachment; and in the Euler tree tour, there are no attachments between the bottom attachment and the top attachment for a leaf bridge. Therefore, we can compute the weights of all leaf bridges by using one All-Prefix-Sums procedure. Similarly, we can compute the top bridge weight. This approach may give us a better implementation in practice.

9.5 Rake and Compress Operations

In this section, we review the Rake and Compress operations that Miller and Reif [MR85,MR87] use for their parallel tree contraction algorithm. Let ***Rake*** be the operation which removes all leaves from a tree T. Let a ***chain*** be a maximal sequence of vertices, $v_1, ..., v_k$ in T, such that v_{i+1} is the only child of v_i for $1 \leq i < k$, and v_k has exactly one child. Let ***Compress*** be the operation that replaces each chain of length k by one of length $\lceil k/2 \rceil$. One possible approach to replacing a chain of length k by one of length $\lceil k/2 \rceil$ is to identify v_i with v_{i+1} for i odd and $1 \leq i < k$.

Let Contract be the simultaneous application of Rake and Compress to the entire tree. Miller and Reif [MR85,MR87] show that the Contract operation need be executed only $O(log\ n)$ times to reduce T to its *mot*. They prove the following theorem.

Theorem 9.5.1 *After $\lceil log_{5/4} n \rceil$ executions of Contract are performed, a tree of* n *vertices is reduced to its mot.*

In this section, we present the important definitions, used in the proof of Theorem 9.5.1, that are needed later on in this paper.

Let V_0 be the leaves of T, V_1 be the vertices of T with only one child, and V_2 be those vertices of T with two or more children. Next, partition the set V_1 into C_0, C_1, and C_2 according to whether its child is in V_0, V_1, or V_2, respectively. Similarly, partition the vertices C_1 into GC_0, GC_1, and GC_2, according to whether the grandchild is in V_0, V_1, or V_2, respectively. Let $Ra = V_0 \cup V_2 \cup C_0 \cup C_2 \cup GC_0$ and $Com = V - Ra$.

Miller and Reif [MR85] show that Rake reduces the size of *Ra* by at least a factor of 1/5 in each application, while Compress reduces the size of *Com* by a factor of 1/2 in each application. We use similar techniques in this paper.

9.6 Isolation and Deterministic EREW Tree Contraction

In Section 9.4 we showed that if we find an EREW parallel tree contraction algorithm which takes $O(\log n)$ time and uses n processors, then we get an $O(\log n)$ time, $n/\log n$ processor, EREW PRAM algorithm for parallel tree contraction. Thus, we may restrict our attention to $O(n)$ processor algorithms. In this section, a technique called isolation is presented and used to implement parallel tree contraction on an EREW PRAM without increasing time and processor count. We also present a method for prudent tree contraction which is important in more general tree problems (for example, see Miller and Teng [MT87]).

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

Figure 9.5: Isolate and Compress for Deterministic Parallel Tree Contraction when $P = n$.

It is important to understand why the deterministic parallel tree contraction, presented by Miller and Reif [MR85,MR87], does not work on the EREW model. The problem arises at the parent v of a chain (a node in V_2 with a child in V_1 is called the **parent** of a chain). Using the pointer-jumping algorithm of Wyllie [Wyl79], we encounter the problem that, over time, many nodes may eventually point to this parent. Now, if v becomes a node in V_1, then all these nodes must determine the parent of v and point to it, which seems to require a concurrent read. We circumvent this problem by isolating the chain until it is Compressed to a point. At that point, we then let it participate in another chain. (See Isolate and Compress presented in Subsection 9.6.1.)

Theorem 9.6.1 (Main Theorem) *Tree contraction can be performed, deterministically, in* $O(n/P)$ *time using* P *processors on an EREW PRAM for all* $P \leq n/\log n$.

9.6.1 Isolation and Local Compress

Figure 9.5 displays a high level description of our deterministic algorithm for parallel tree contraction on an EREW PRAM which uses $O(\log n)$ time and n processors which we call **Isolate and Compress.**

The difference between the contraction phase used in this algorithm and the dynamic contraction phase presented by Miller and Reif [MR85,MR87] is that the **Compress** operation here is replaced by two operations: Isolation and Local Compress. Each Local Compress operation applies one conventional Compress operation to an isolated chain during each contraction phase.

Lemma 9.6.1 After *each Isolate and Compress* $|Com|$ *decreases by a factor of at least 1/4.*

Proof: By the way the steps Isolation and Integration are implemented, each isolated chain has a length of at least 2. Moreover, no two consecutive nodes are singletons. Thus, a chain consists of an alternating sequence of an isolated chain and a singleton. We view a chain as a set of pairs where each pair consists of an isolated chain and a singleton. Each pair is decreased in size by at least a factor of 1/4. The worst case is a pair containing an isolated chain of size 3. □

Since step Rake removes 1/5 of Ra and steps Isolation through Intergration removes 1/4 of Corn, together they must remove 1/5 of the vertices. This gives the following theorem.

Theorem 9.6.2 *A tree of n vertices is reduced to its root after one applies Isolate and Compress* $\lceil \log_{5/4} n \rceil$ *times.*

The next two sections present a more detailed implementation of Isolate and Compress.

9.6.2 Implementation Techniques

In this subsection, we present one method of implementing the generic contraction phase on an EREW model in $O(\log n)$ time, using n processors. Another implementation method is presented in the next subsection. The complexity of these two implementation methods differs only by a constant factor. However, they have different scopes of application.

We view each vertex, which is not a leaf, as a function to be computed where the children supply the arguments. For each vertex v with children, $v_1 \ldots v_k$, we will set aside k locations, $l_1 \ldots l_k$, in common memory. Initially, each l_i is empty or **unmarked.** When the value of v_i is known, we assign it to l_i and denote it by **mark** l_i. Let $Arg(v)$ denote the number of unmarked l_i. Then, initially, $Arg(v) = k$, the number of children of v. We need one further notation: Let **vertex (P(v))** be the vertex associated with the sole parent of v with storage location $P(v)$. All vertices shall be **tagged** with one of four possibilities: G, M, R, or 8. 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 $Tag(v) = 8$; otherwise, v is **not free.**

To determine whether or not a child is free, we assign each vertex a new variable that is read only by the parent. To determine if a parent is free, we require that each vertex keep a copy of this variable for each child. Initially, all copies indicate that the vertex is not free. When the parent vertex becomes free, it need only change the copy of the variable associated with the remaining child, since there are no other children that can read the other variables.

```
Procedure ISOLATE-COMPRESS
  In parallel, for all v ∈ V − {r}, do
    case Arg(v) equals
    0) Mark P(v) and delete v ;;
    1) case Tag(v) equals
       ∅) if child is not free and parent is free then Tag(v) ← G
          if parent is not free and child is free then Tag(v) ← R
          if parent is free and child is free then Tag(v) ← M ;;
       G) if Tag(P(v)) = R then Tag(v) ← ∅
          P(v) ← P(P(v)) ;;
       M) if Tag(P(v)) = R then Tag(v) ← R
          P(v) ← P(P(v)) ;;
       esac
    esac
  od
```

Figure 9.6: The First Implementation of the *ISOLATE-COMPRESS* Procedure.

9.6.3 The Expansion Phase

If we use procedure *ISOLATE-COMPRESS* to, say, evaluate an arithmetic expression it will not return the value of all subexpression. There is one a one-to-one correspondence between subexpression and subtrees of the expression tree. For many application it is necessary to compute the value of all subexpressions. This is usually done by running the contraction phase "backwards" which is called parallel tree expansion, see [MR87]. To insure that the expansion phase only uses exclusive reads we must be a little careful, since many nodes may need the value of the same node in order to compute its value. Thus one solution requires each node to maintain a queue of $O(\log n)$ pointers. We store one pointer at node v for each node w which needs the value of v to determine its value. A solution using only a constant amount of space per node can be achieved by several methods.

One easy to describe method can be obtained by reversing the pointers in each isolated chain and compressing these chain based one the reverse pointers. In this case, the original root is now the tail and is tagged G while the original tail is now the root and is tagged R. Otherwise, we run Procedure *ISOLATE-COMPRESS* as in Figure 9.6. Each time we apply the procedure we will increment a counter which we think of as the time. When a node v obtains a R tag (except the' new root node of the chain) it records the time (the value of the counter) using variable t_v and the node n_v with tag R that it is now pointing. In the expansion phase node v determines the value of node n_v at time t_v with the clock running "backwards".

One can also perform the expansion phase without reversing the pointers in an isolated chain by simulating the above method directly on the original forward pointers. In this case, during the contraction phase if a node v is pointing to a node w and values have been assigned to the variables t_v and n_v or the tag of v is G then node v sets t_w equal to the value of the counter and n_w to w. This method is basically the same as the first method. The expansion is the same as in the first method.

9.6.4 Prudent Parallel Tree Contraction

One disadvantage of the Local Compress procedure is that, during each Compression stage, one useless chain is produced out of each chain. For generic tree contraction this disadvantage appears harmless; but for more general tree problems, such as those involved in evaluating min-max-plus-times-division trees where the cost to represent the function on each edge doubles with each functional composition, a factor of n is added to the number of processors used, and a factor of logn is added to the running time [MT87].

The Compress procedure, where no useless chain is produced, is called **prudent Compress.** Using prudent Compress, as much as a factor of n may be saved in the number of processors used in certain applications (see Miller and Teng [MT87] for further details).

Our idea is very simple: first, isolate each chain; second, use Wyllie's [Wyl79] List-Ranking algorithm to rank the vertices in the chain; and third, use this ranking of the vertices in the chain to determine the order in which pairs of vertices are identified. The procedure is written assuming that the parallel tree contraction is run asynchronously: i.e., a block of processors and memory are assigned the task of evaluating a subprocedure which is then performed independently of the rest of the processors and memory. It is unnecessary to write the procedure this way, but it is easier to follow. The Compress part of the procedure is written; the Rake part remains the same.

Procedure *PRUDENT-COMPRESS*

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 which computes the ranks of all nodes in an isolated chain C and uses this information to Compress the chain in such a way that

time, then a natural modification of parallel tree contraction, called Asynchronous Parallel Tree Contraction, contracts the tree in $O(\log n)$ time. These techniques work with the Isolation techniques for Compress.

The second way of performing the Rake operation for trees of unbounded degree is to identify consecutive pairs of leaf siblings as we did when we used Compress for parent-child pairs [MR]. A run of leaves is a maximal sequence of leaves, $l_1, \ldots, l_k$, which are consecutive siblings. We assume that the siblings are cyclically ordered (i.e., the left-most child follows the right-most child). The operation, **Rake Restricted to Runs,** replaces each run of length k by one of length $\lceil k/2 \rceil$ for $k \geq 2$; and, any run of length 1 is removed completely. We also remove both leaves when they are the only siblings.

Theorem 9.6.4 *Pamllel Tree Contraction, where Rake is restricted to runs and Compress uses Isolation, reduces a tree of size n to its root in* $\lceil \log_{7/6} n \rceil$ *applications.*

Proof: The proof is a straight forward calculation based upon techniques presented by Miller and Reif [MR87,MR].

References

[AM87] Richard Anderson and Gary L. Miller. ***Optimal Parallel Algorithms for List Ranking.*** Technical Report , USC, Los Angeles, 1987.

[BV85] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree form. ***ACM Transactions on Programming Languages and Systems,*** 7(2):348–357, April 1985.

[CV86a] 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.

[CV86b] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal list ranking. ***Information and Control, 70***(1):32–53, 1986.

[DNP86] Eliezer Dekel, Simeon Ntafos, and Shie-Tung Peng. ***Parallel Tree Techniques and Code Opimization,*** pages 205–216. Volume 227 of ***Lecture Notes in Computer Science,*** Springer-Verlag, 1986.

[KU86] Anna Karlin and Eli 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.

[MR] Gary L. Miller and John H. Reif. Parallel tree contraction part 2: further applications. *SIAM J. Comput.* submitted.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science,* pages 478–489, IEEE, Portland, Oregon, 1985.

[MR87] Gary L. Miller and John H. Reif. *Parallel Tree Contraction Part 1: Fundamentals.* Volume 5, JAI Press, 1987. to appear.

[MT87] Gary L. Miller and Shang-Hua Teng. Systematic methods for tree based parallel algorithm development. In *Second International Conference on Supercomputing,* pages 392–403, Santa Clara, May 1987.

[Ran87] A. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science,* pages 185–194, IEEE, Los Angeles, Oct 1987.

[TV85] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Cornput.,* 14(4):862–874, November 1985.

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computation.* Technical Report TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.