PARALLEL TREE CONTRACTION'

PART 1: FUNDAMENTALS

Gary L. Miller and John H. Reif

ABSTRACT

This paper introduces parallel tree contraction: a new bottom-up technique for constructing parallel algorithms on trees. Contraction can be used to solve a wide variety of problems. Two examples included in this article are expression evaluation and subexpression elimination. In this paper we show these applications only require $O(\log n)$ time and $O(n/\log n)$ processors on a 0-sided randomized **PRAM** or $O(n)$ processors on a deterministic **PRAM.** In the process of finding these efficient algorithms we find efficient parallel algorithms for several other problems including generating random permutation in parallel and randomized techniques for work load balancing on PRAMs.

We have found other application of parallel tree contraction including testing isomorphism of trees, canonical forms for trees,

Advances in Computing Research, Volume 5. **pages** 47–72.

ISBN: **0–89232–896–7**

constructing planar embeddings, and testing isomorphism of planar graphs. These applications appear in a companion paper [MR2].

1. INTRODUCTION

1.1. Top-Down vs Bottom-Up Tree Algorithms

Trees play a fundamental role in many computations, both for sequential as well as parallel problems. The classic paradigm applied to generate parallel algorithms in the presence of trees has been divide-conquer; finding a vertex which is a **"1/3–2/3"** separator, and recursively solving the two subproblems. A now classic example is Brent's work on parallel evaluation of arithmetic expressions [B]. This "top-down" approach has several complications, one of which is finding the separators that must separate the tree into components with size $\leqslant 2/3$ of the original size. We define dynamic expression evaluation as the task of evaluating the expression with no free preprocessing. If we apply Brent's method, finding the separators seem to add a factor of logn to the running time.

We give a "bottom-up" algorithm to handle trees. That is, all modifications to the tree are done locally. This "bottom-up" approach, which we call ***CONTRACT,*** has two major advantages over the "top-down" approach: (1) The control structure is straightforward and easier to implement facilitating new algorithms using fewer processors and less time. **(2)** Problems for which it was too difficult or too complicated to find polylog parallel algorithms are now easy. We believe our lasting contribution will be CONTRACT. It has already been applied to finding small separators for planar graphs in parallel [M] as well as numerous applications appearing in the companion paper [MR2].

1.2. The PRAM Model

We shall use the ***PRAM*** model of a parallel processing device (see [SV]). A PRAM consists of a collection of processors. Each processor is a random access machine that can read and write in a common random access memory. In unit time these processors are allowed concurrent reads and concurrent writes (CRCW), as well as arithmetic operations on integers of magnitude upper bounded by $n^{O(1)}$. There are two natural implementations of concurrent writes. (1) If

two or more processors attempt to write in a given location of common memory then one of the processors will succeed. The performance of the algorithm should not depend on which processor succeeds. (2) In the second model concurrent writes in a given location cause detectable noise to be stored in that location. Unless otherwise stated we shall assume the first model for concurrent writes. But all of our algorithms also work with the same performance in the second model.

Many of our algorithms use randomization. That is, each processor has access to an independent random number of magnitude $< n$ per step. A (1-sided) randomized algorithm A is said to accept a language L in $T(n)$ time using $P(n)$ processors if the following conditions hold: (1) on all inputs w of length n, A uses at most $T(n)$ time and $P(n)$ processors independent of the random bits; **(2)** if A rejects w then $w \notin L$; (3) if A accepts w then with probability of error at most $1/n$ we can conclude that $w \in L$. Note that we have chosen $1/n$ for our error bound instead of the common value $1/2$. It generally seems to increase the running time by a factor of log n to achieve the error bound $1/n$ from an algorithm with error bound $1/2$. On the other hand, given an algorithm with error bounded by $1/n$ if we increase the running time by a constant factor of a we can achieve the tighter error bound $1/n^{\alpha}$. We say an algorithm is 0-sided randomized if it is always correct when it terminates and the probability of termination is at least $1 - 1/n$. We often denote 0-sided and 1-sided by subscripts of **0** and 1 respectively (see [R1]).

All our PRAM algorithms will only use a polynomial number of processors. We shall take considerable effort to minimize the number of processors used.

1.3. Expression Evaluation and Our Results

Arithmetic expression evaluation is a good robust problem exhibiting our techniques. An arithmetic expression is a tree where the leaves have values from some domain and each internal vertex has two children and a label from $\{+, \times, \div\}$. We assume that these binary operations can be performed in constant time. As we shall see in the companion paper, these techniques are very general but most of the ideas will be well illustrated in the case of expression evaluation.

We exhibit a deterministic **PRAM** algorithm for dynamic expression evaluation using $O(\log n)$ time and $O(n)$ processors and a 0-sided randomized version of this algorithm using only $O(n/\log n)$ processors. We then extend these algorithms to evaluate all subexpressions using the same time and number of processors. In comparison Brent [B] showed that expressions of size n could be transformed into straight-line code of depth $O(\log n)$. Dynamic transformation of code in parallel by Brent's method seems to require $\Omega(\log^2 n)$ time.

We also give an algorithm that uses the same resource bounds as the expression case for computing the value of all subexpressions. This result is a natural generalization of parallel prefix evaluation [F, LF, V]. Up to constant factors we use no more time or processors. The list-ranking problem (see [V]) is also a special case. Thus, we have given an $O(\log n)$ time optimal algorithm for list-ranking. This is the first known $O(\log n)$ time and $n/\log n$ processor algorithm for the problem.

1.4. Organization of This Paper

The body of the paper consists of 6 sections. In Section 2 we define two abstract operations on trees, RAKE and **COMPRESS.** We show that only $O(\log n)$ simultaneous applications of these operations are needed to reduce a tree to a point. In Section 3 we give both deterministic as well as randomized implementations **of RAKE** and **COMPRESS** both of these implementations reduce a tree to a point in $O(\log n)$ time using $O(n)$ processors. In Section **4 we show how** to implement these operations on a randomized **PRAM** such that any tree is reduced to a point in $O(\log n)$ time using an optimal number of processors. We call this implementation dynamic tree contraction. In Section **4** we also describe how to generate a random permutation in $O(\log n)$ time using only $n/\log n$ processors and how to remove a constant proportion of the zeros from a random string in $O[\log(\log n)]$ time using only $n/\log(\log n)$ processors. These operations give a general technique for minimizing the number of processors used. We call this general technique processor work load balancing. In Sections **5** we apply dynamic tree contraction to expression evaluation, subexpression evaluation and related tree problems. In Section 6 we show that a natural modification of parallel tree contraction, which we call asynchronous parallel tree contraction, works in $O(\log n)$ time even

if the cost of raking k siblings is $O(\log k)$ time. In Section 7 we give a partial analysis of the random variable ***MATE*** that arises in the randomized parallel tree contraction algorithms.

2. THE RAKE AND COMPRESS OPERATIONS

Let $T = (V, E)$ be a rooted tree with n vertices and root r. We describe two simple parallel operations on T such that at most $O(\log n)$ applications are needed to reduce T to a single vertex.

Let ***RAKE*** be the operation that removes all leaves from T. It is easy to see that **RAKE** may need to be applied a linear number of times to a highly unbalanced tree to reduce T to a single vertex. We can circumvent this problem by adding one more operation.

We say a sequence of vertices $v_1, \ldots, v_k$ is a ***chain*** if v_{i+1} is the only child of v_i for $1 \leqslant i \leqslant k$, and v_k has exactly one child and that child is not a leaf. The chain is said to have length k. In one parallel step, we ***compress*** a chain by identifying v_i with v_{i+1} for i odd and $1 \leqslant i < k$. Thus, the chain $v_1, \ldots, v_k$ is replaced with a chain $v_1', \ldots, v_{\lfloor k/2 \rfloor}'$. Let ***COMPRESS*** be the operation on T that "compresses" all maximal chains of T in one step. Note that a maximal chain of length one is not affected by **COMPRESS.**

Let ***CONTRACT*** be the simultaneous application of **RAKE** and **COMPRESS** to the entire tree. We next show that the **CONTRACT** operation needs only be executed $O(\log n)$ times to reduce T to ***its root.***

THEOREM 2.1. ***After*** $\lfloor \log_{5/4} n \rfloor$ ***applications of CONTRACT to a tree on n vertices it is reduced to its root.***

Proof. We partition the vertices of T into two sets ***Ra*** and ***Com*** such that $|Ra|$ will decrease by a factor of **4/5** after an execution of RAKE and $|Com|$ will decrease by a factor of 1/2 after **COMPRESS.**

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

To see that the size of ***Ra*** decreases by a factor of 1/5 after each RAKE we show that $|Ra| \leq 5|V_0|$. The inequality follows by observing the following inequalities: $|V_2| < |V_0|$, $|C_0| \leq |V_0|$, $|GC_0| \leq |V_0|$, and $|C_2| \leq V_2|$.

Note that all vertices in V_1 except those of C_0 belong to a chain. Thus, every vertex of ***Corn*** belongs to some maximal chain. If $v_1, \ldots, v_k$ are the vertices of a maximal chain then either $v_k \in C_2$ or $v_k \in GC_1$. In either case $v_1, \ldots, v_{k-1}$ are the only elements in the chain belonging to ***Corn.*** Thus, the number of elements in a maximal chain of ***Corn*** decreases by at least a factor of 1/2 after COMPRESS. □

The type of argument used in the proof of Theorem 2.1 will be used in the analysis of several other algorithms which are based on CONTRACT. Given a tree $T = (V, E)$ let $Rake(V) = Ra$ and $Compress(V) = Corn$ as defined in the above proof.

There are many useful applications of parallel tree contraction and expansion. For each given application, we associate a certain procedure with each RAKE and COMPRESS operation that we assume can be computed in parallel quickly. Typically the vertices of the tree Twill contain labels storing information relevant to the given application. The RAKE and COMPRESS operations will modify these labels, as well as the tree itself. To apply parallel tree contraction to a problem seems to require finding a general form for implementing and storing the composition of unary functions.

As a simple example we consider the case when T is an expression tree over $\{+, \times\}$ the RAKE corresponds to the operation of (1) evaluating a vertex if all of its children have been evaluated or (2) partially evaluating a vertex if some of its children have been evaluated. The cost of applying RAKE to an expression tree is the cost of evaluating a vertex. If a vertex has been partially evaluated except for one child then the value of the vertex is a ***linear function,*** say, $aX + b$ where Xis a variable of the remaining child, and a and b are scalars over some semiring. Thus a chain is a sequence of vertices each of which is a linear function of its child. In this application, COMPRESS is simply pairwise composition of linear functions. Thus, in this example the only nontrivial observation is the fact that linear functions in one argument are closed under composition and each linear function can be represented by two scalars.

This gives a simple proof that (after preprocessing) expressions can be evaluated in time $O(\log n)$ using $O(n)$ processors on a **PRAM.** On the other hand, the naive dynamic implementation of **COMPRESS** requires $O(\log n)$ time since we first will determine the parity of each vertex on a chain by pointer jumping, e.g. (doubling-up), then combine consecutively the odd and even vertices pairwise in constant time. In the next section we implement both a deterministic and a randomized variant of **COMPRESS** that can be performed in constant time.

3. DYNAMIC TREE CONTRACTION (DETERMINISTIC AND RANDOMIZED)

In this section we describe in more detail two implementations of **COMPRESS.** The first is deterministic while the second is a randomized algorithm (see Section **3.2).** The deterministic algorithm seems to need $O(n)$ processors to achieve $O(\log n)$ time. We will show in Section 4 how to improve the randomized algorithm to use only $O(n/\log n)$ processors and $O(\log n)$ time. In this section we assume that the trees are of bounded degree. The analysis of parallel tree contraction on trees of unbounded degree is in Section **6.**

3.1. Deterministic Tree Contraction

Let T be a rooted tree with vertex set V of size $n = |V|$ and root $r \in V$. 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 will assign it to l_i : this will be simply denoted by mark l_i. Let $Arg(v)$ denote the number of unmarked l_i. Thus, 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 $P(v)$. Figure 1 contains a single phase of procedure dynamic contraction.

The procedure must detect when $Arg(v)$ equals 0 or 1. If the number of arguments per vertex is bounded this can be tested in constant time using the processor assigned to vertex v. In the case when the number **of** arguments is unbounded we can assign **a**

Figure 1. A dynamic contraction phase.

Procedure `Dynamic Tree` Contraction:
In Parallel for all $v \in V - \{r\}$ `do`
1) **If** $Arg(v) = 0$ **then** `mark` $P(v)$ **and** delete v;
2) **If** $Arg(v) = Arg(vertex(P(v))) = 1$ **then** $P(v) \leftarrow P(vertex(P(v)))$.
od

processor to each argument still using at most $O(n)$ processors since the total number of arguments is n − 1. These processors assigned to the arguments of v can test whether $Arg(v) = 0$ by having each processor Q without an argument perform a concurrent write of its index into some memory location m_v of v. $Arg(v) > 0$ if and only if some index is written into this memory location. To further test if $Arg(v) > 1$, we have each processor Q with no argument read m_v and if the value is not the index of Q then Q again writes its index in m_v. Thus, $Arg(v) > 1$ if the value of m_v changes on the second write. In Section 6 we show that procedures that takes at most $O(\log k)$ to test $Arg(v)$ equal 0 or 1 will still give an overall running time of $O(\log n)$, where k is the number of arguments of v.

The procedure implements the RAKE in the straightforward way, while the operation **COMPRESS** is implemented by pointer jumping. In line (2) of the procedure each vertex in a chain adjusts its pointer P, which was initially pointing at its parent, to point at its grandparent.

More intuition for the procedure dynamic contraction can be gained by seeing it applied to expression evaluation over $\{+, \times\}$. If $Arg(v) = 0$ then v "knows" its value and passes it on to its parent. We can test if $Arg(v) = 0$ or $Arg(v) = 1$ in constant time using concurrent reads and writes. If v and $P(v)$ are functions of one remaining argument we will view them as linear functions of their argument. We store these functions in common memory indexed by the corresponding vertex. Thus v reads the linear functions of $P(v)$, composes it with its own function, and adjusts its pointer to $P\{vertex[P(v)]\}$ It follows that this correctly computes the value of the expression. We next analyze the number of applications of dynamic contraction.

THEOREM 3.1. *The number of applications of dynamic tree contraction needed to reduce a tree of n vertices to its root is bounded above by the* number for *CONTRACT.*

Proof. Observe that every maximal chain, after dynamic tree contraction, decomposes into two chains, one ***essential chain*** corresponding to COMPRESS and an unnecessary chain that is out of phase. This second chain has a leaf that is unevaluated. For the purpose of analysis we can discard the second chain from the analysis since it will never be evaluated. Thus, a single phase of dynamic tree contraction is just CONTRACT, after discarding the unevaluatable chains. It is important to point out that dynamic tree contraction is slightly faster than CONTRACT since it does not test if the only child of a vertex is a leaf or not. Thus, some pointer jumping occurs in dynamic tree contraction that does not occur in CONTRACT. We used the more conservative contract in CONTRACT since we felt that for many applications a vertex with an only child will use the time at this stage to evaluate itself rather than pointer jumping. □

Note that many vertices are not evaluated, that is, for many vertices v the value $Arg(v)$ is never set to **0** during any stage of dynamic tree contraction. We will define a new procedure dynamic tree expansion that will allow the evaluation of all vertices, i.e., each vertex will eventually have all its arguments after completion of the procedure. We modify dynamic tree contraction so that each vertex keeps a push-down store *Store,* of all the previous values of $P(v)$. Here we add line **(0)** at the start of the block inside the ***do*** and ***od*** of dynamic tree contraction.

0) Push on *Store,* value $P(v)$.

We now apply dynamic tree contraction until the root r has all its arguments. Next we apply procedure dynamic tree expansion given in Figure 2 until all vertices have all their arguments.

We must show that after successive applications of dynamic tree expansion all vertices have their arguments. **As** in the proof of Theorem **3.1** we can discard those chains that have a leaf that will not be evaluated. The proof is by induction on the trees with only

Figure 2. **A** dynamic expansion phase.

Procedure Dynamic Tree **Expansion:**
  In **Parallel for** all $v \in V - \{r\}$ **do**
    1) $P(v) \leftarrow Pop(Store_v)$:
    2) **if** $Arg(v) = 0$ **then** mark $P(v)$.
  od

essential chains, as defined in the proof of the previous theorem, starting from the trivial tree consisting of a singleton vertex r and finishing with the original tree T, say, $\{r\} = T_1, \ldots, T_k = T$. Now every vertex in T_{i+1} is either a leaf in which case we know its value or this vertex is missing one argument that is the value of a vertex in T_i. In the latter case this value will be supplied in one application of dynamic tree expansion. This gives the following theorem.

THEOREM 3.2. At most $\lfloor \log_{5/4} n \rfloor$ applications of dynamic tree contraction and $\lfloor \log_{5/4} n \rfloor$ applications of dynamic tree expansion are needed to mark all the vertices.

3.2. Randomized Tree Contraction and Expansion

We next describe a randomized version of CONTRACT. This algorithm has the disadvantage that it needs access to many random numbers but it has the advantages that (1) in many cases, it will only use about half as many function evaluations and (2) it can be modified into an algorithm that up to constant factors uses an optimal number $O(n/\log n)$ of processors and still runs in time $O(\log n)$. We describe the algorithm in procedure form (see Figure 3).

The rest of this section contains a probablistic analysis of the procedure randomized contract. We believe that good analysis of this procedure with attention to constants is important. We first show that roughly 1/5 of the vertices are deleted with probability at least 1/2. We use this bounded to show that randomized contract will reduce a tree to a single vertex in $O(\log n)$ time with high probability. For the processor efficient randomized contraction algorithms presented in Section 4.4 we need that randomized contraction deletes a constant proportion of the vertices with high

Figure 3. **A RANDOMIZED CONTRACT phase.**

Procedure RANDOMIZED CONTRACT:
In Parallel for all $v \in V - \{r\}$ **which** have not been deleted **do**
1) **If** $Arg(v) = 0$ **then mark** $P(v)$ and delete v;
2) **If** $Arg(v) = 1$ **then** Randomly **assign** M or F to Sex(v);
3) **If** $Sex(v) = F$ and $Sex(P(v)) = M$ **then do**
a) Push on $Store_v$ **value** $P(v)$;
b) $P(v) \leftarrow P(P(v))$;
c) **delete vertex**$(P(v))$.
od
od

probability for n large. Thus, we show that randomized contract deletes at least $n/32$ vertices with probability of failure at most $1/n$. Note that if we are not concerned about constants then the second analysis would suffice for both applications.

The analysis will follow arguments similar to those used in the proof of Theorem 2.1. Here we partition the vertex set V into Rake(V) and Compress(V) as defined in that proof. Again by similar argument step (1) of RANDOMIZED CONTRACT will delete at least 1/5 of the vertices in Rake(V). Steps (2) and (3) of randomized CONTRACT are called ***Randomized Pointer Jumping.*** The **expected number** of **vertices** of **Compress(V) that are deleted** in step (3c) is $m/4$ where $m = |Compress(V)|$. We cannot directly conclude that the median is also $m/4$. Recall, that the ***median*** of a random variable X is the maximum real number $\mu(X)$ such that $Prob[X \leqslant \mu(X)] \leqslant 1/2$. We can lower bound the median using the expected number and the variance of the number of vertices deleted. Since the number of deleted vertices in each maximal chain is mutually independent, the number of deleted vertices is the sum of independent random variables, one for each maximal chain. Let $C_1, \ldots, C_k$ be a list of maximal chains in T where C_i is a chain of length $m_i + 1$. Thus, m_i of the vertices of C_i are members of the set Compress(V). Let the number of deleted vertices in the chain C_i after one application of RANDOMIZED CONTRACT be the random variable $MATE_{m_i}$. If $m = |Compress(V)|$ then the random variable that is the number of deleted vertices in one phase will be $X = MATE_{m_1} + \cdots + MATE_{m_k}$ where k is the number of maximal chains. Thus, the expected value of X is $E(X) = m/4$. By Lemma 7.1 the variance for one chain is $(m_i + 2)/16$. Thus, the variance for X is $\sum_{i=1}^{k} (m_i + 2)/16 = (m + 2k)/16$. The variance is maximized when each $m_i = 1$. In this case the variance is $Var(X) = 3m/16$. The Chebyshev's inequality gives the following estimate for the median of X, $\mu(X)$ (see **[L], p.** 244).

LEMMA 3.3. $|\mu(X) - E(X)| \leqslant \sqrt{2\,Var(X)}$.

Thus $\mu(X) \geqslant E(X) - \sqrt{2\,Var(X)}$. In our case this gives $\mu(X) \geqslant m/4 - \sqrt{3m/8}$. Therefore for sufficiently large m, $m \geqslant 150$, $\mu(X) \geqslant m/5$. After some simple computer calculations we conjecture that $\mu(X) \geqslant m/5$ for $m \geqslant 15$ (see Section 7).

THEOREM 3.4. ***RANDOMIZED CONTRACT deletes at least*** $n/5 - 150$ ***vertices with probability at least*** $1/2$.

Proof. Let T be the tree input to randomized contraction and $m = |Compress(V)|$. Thus, $n - m = |Rake(v)|$. We know that at least $(n - m)/5$ vertices in $Rake(v)$ are deleted in every phase. We know from the last lemma that for $m \geqslant 150$ at least $m/5$ of the vertices in Compress(V) are also deleted with probability 1/2. Thus, $m/5 - 150$ of the vertices in $Compress(V)$ are deleted with probability 1/2. Therefore the total deleted is at least $(n - m)/5 + m/5 - 150 = n/5 - 150$.

Let S_n be the number of successes in n independent trials with probabilityp of success on each trial. We shall need one major fact about the binomial random variable S_n – the probability of being more than any fixed constant factor from the expected value is exponentially small. This fact was observed by Uspensky [U] (see [JK]). These bounds are commonly known as Chernoff bounds [C]. We shall use the following simply stated bounds [AV].

THEOREM 3.5. ***For any*** n, p, ε ***with*** $0 \leqslant p \leqslant 1, 0 \leqslant \varepsilon \leqslant 1$:

$$Prob(S_n \leqslant \lfloor (1 - \varepsilon)np \rfloor) \leqslant e^{-\varepsilon^2 np/2}$$

and

$$Prob(S_n \geqslant \lceil (1 + \varepsilon)np \rceil) \leqslant e^{-\varepsilon^2 np/3}.$$

We use these bounds to show:

THEOREM 3.6. ***After*** $\lceil 12.5 \log n \rceil + 150$ ***applications of RANDOMIZED CONTRACT a tree of n vertices will be reduced to a single vertex with probability offailure at most*** $1/n$.

Proof. We show that after $k = \lceil 12.5 \log n \rceil$ applications of RANDOMIZED CONTRACT a tree of size n is reduced to a tree of size 150. Since randomized contract always removes at least the leaves of a tree the tree of size 150 will take at most 150 more steps. We say a given application of randomized contract is a ***success*** if it deletes $n/5 - 150$ vertices from a tree of size n and a ***failure*** otherwise. If after k applications the tree has not been reduced to one of size 150 then we must have had less than $\lceil \log_{5/4} n \rceil$ successes. In Lemma 3.4 we showed that probability of success was at least 1/2 independent of the tree. Thus the probability that the tree has more than 150 vertices after k application is bounded by $Prob(S_k \leqslant \lfloor \log_{5/4} n \rfloor)$ where $p = 1/2$. We use the first inequality from Theorem 3.5 to bound this probability. We set $n = k, \varepsilon = 1/2$, and $p = 1/2$

and check that $\lfloor \log_{5/4} n \rfloor \leqslant \lfloor (1-\varepsilon)kp \rfloor$. The last inequality is a straightforward calculation. Thus the probability of failure after k applications is at most $e^{-\varepsilon^2 kp/2}$. To get a probability of error at most $1/n$ we must just see that $\varepsilon^2 kp/2 \geqslant \ln n$. That is $k \geqslant 16 \ln n$, but $16 \ln n \leqslant 12 \log n \leqslant k$. □

We next show that RANDOMIZED CONTRACT will delete at least $n/32$ vertices with only vanishingly small probability of failure.

THEOREM 3.7. ***One phase of RANDOMZZED CONTRACT for any*** $n \geqslant 180$ ***will delete at least*** $n/32$ ***vertices with a probability of failure less than*** $1/n$.

Proof. Let n be the number of vertices in a tree T and m be the number of vertices in Compress (T). If $m \leqslant 27n/32$ then $n - m \geqslant 5n/32$ vertices are in Rake (T) and therefore at least $1/5(5n/32) = n/32$ of them are deleted by RAKE. In this case $n/32$ of the vertices are deleted by RAKE alone without considering vertices deleted by COMPRESS. Thus, we may assume that $m > 27n/32$. It will suffice to show that $m/32$ of the vertices in Compress (T) are deleted by RANDOMIZED CONTRACT with small probability of failure. Let $Z \subset$ Compress (V) be a maximum subset of vertices such that no vertex in Z is a parent of another vertex in Z, i.e. Z is a maximal independent set. Now each vertex in Z is deleted independently with probability $1/4$. Since the induced graph on Compress (T) is a forest, the number of vertices in $|I| \geqslant \lceil m/2 \rceil$. Thus, the number of vertices deleted is bounded below by the binomial random variable $S_{\lceil m/2 \rceil}$ where p $= 1/4$. The probability of failure is bounded by

$$\text{Prob}\,(S_{\lceil m/2 \rceil} \leqslant \lfloor m/32 \rfloor) \leqslant \text{Prob}\,(S_{\lceil m/2 \rceil} \leqslant \lfloor (1-\varepsilon)\lceil m/2 \rceil/4 \rfloor),$$

where $\varepsilon = 3/4$. Using the Chernoff bounds from Theorem 3.5 this probability at most

$$\leqslant e^{-\varepsilon^2 \lceil m/2 \rceil/8} \leqslant e^{-\varepsilon^2 m/16}.$$

Using the hypothesis that m $\geqslant 27n/32$ and $\varepsilon = 3/4$ we get the above probability

$$\leqslant e^{-(3/4)^2(27/32)n/16} = e^{-(3^5/2^{13})n} \leqslant e^{-n/34}.$$

For $n \geqslant 180$ we have that $e^{-n/34} \leqslant 1/n$. □

4. AN OPTIMAL RANDOMIZED TREE EVALUATION ALGORITHM

4.1. Improving the Processor Count by Load Balancing

In this section we show how to implement RANDOMIZED CONTRACT on a tree T with n vertices so that T is reduced to its root in $O(\log n)$ time using $O(n/\log n)$ processors. We will contract T to a tree T' of size $n/\log n$ at which point we will have a tree small enough so there is a processor for every vertex of T' and can use one the deterministic algorithm from the last section. The important difference in the reduction is that we will be operating on an array of n vertices using only $o(n)$ processors as opposed to one processor for each pointer value. We consider pointers to be either dead or alive. If all pointers of the array are alive and we have p processors then we simply assign intervals of pointer values of size $\lceil n/p \rceil$ to a single processor.

If the live pointers are interspersed with dead pointers then the time required for some particular processor to finish its tasks may be much longer than the expected or average time. We give a method of balancing the work load using randomization. We consider the processors to be numbered consecutively. In general if A is an algorithm originally specified using p processors but only p' are available we will assume that A is implemented by assigning each distinct interval of $\lceil p/p' \rceil$ virtual processors to one actual processor.

Note that by Theorem 3.7 after each phase of randomized contract with very high probability at least 1/32th of the processors are assigned to dead pointers. Thus, after $O[\log(\log n)]$ phases we will have only $n/\log n$ active processors. One can assign active tasks to an initial sequence of processors by computing all prefix sums as follows.

Let $s_1 \ldots s_n$ be a sequence of zeros and ones where $s_i = 1$ if processor i is active and $s_i = 0$ otherwise, and $a_k = \sum_{i=1}^{k} s_i$. We now assign the task of processor i to processor a_i. It is well known, see [LF]:

LEMMA 4.1. All *prefix sums of* a string *of* length n can be computed in $O(\log n)$ time using $n/\log n$ processors.

This motivates a simple randomized tree evaluation algorithm using $O(n \log(\log n)/\log n)$ processors and $O(\log n)$ time (see Figure 4).

Figure *4.* **A** Randomized tree evaluation (simple form).

Procedure Randomized **Tree** Evaluation (Simple form):
1) $p \leftarrow \lceil n \log(\log n)/\log n \rceil$; $k \leftarrow 1$;
2) While k $\leq c(\log(\log n))$ **do**
T $\leftarrow$ Randomized Contraction(T); (using p processors) (*)
3) Using all prefix sums calculation assign the active tasks to an initial sequence of processors;
4) While $|T| > 1$ **do** T $\leftarrow$ RANDOMIZED CONTRACT(T)

To see that it works in $O(\log n)$ time we use Theorem **3.7.** Note that for some constant c and large enough n, step (2) will reduce T to a tree on $\lceil n/\log n \rceil$ vertices with probability of failure $\leqslant 1/n$. Now each execution of (*) will take $O[\log n/\log(\log n)]$ time. Thus, step (2) requires $O(\log n)$ time. **By** Lemma 4.1 step **(3)** takes only $O(\log n)$ time. **By** the first remark and large enough c we have $|T| \leqslant n/\log n$. Thus, step **(4)** will only take $O(\log n)$ time with probability of failure $\leqslant 1/n$.

Thus, the simple form of randomized tree evaluation reduces the processor count to O[n log (log$n$)/log n], by only "load balancing" once. To remove the last log(log$n$) factor we will load balance between each application of (*). The goal will be to partially balance the load as opposed to performing the balancing exactly. We do the partial balancing by first randomly permuting the tasks and next partially balancing the almost random string of tasks.

4.2. Generating a Random Permutation

In this section we give a processor efficient algorithm to generate random permutations. Another algorithm appears in [R2]. In particular we show:

THEOREM 4.2. There exists *a* randomized ***PRAM*** algorithm that generates random permutations *of* n cells using $O(\log n)$ time, $O(n/\log n)$ processors, and probability *of failure is* at most $1/n$.

The idea behind the algorithm is extremely simple. We shall randomly assign the n cells among 2n cells. We call the assigned position an accommodation. Next we remove the unused cells using prefix calculations as described in the previous section. To get the original assignment of the n cells into 2n we will require each **of** the $n/\log n$ processors to be responsible for finding accommodations for

$\log n$ cells. Each processor starts at the beginning of its list of cells and chooses a random accommodation. The processor will find an accommodation for the cell with probability at least $1/2$. Thus, the expected completion time for each processor is at most $2 \log n$. We allow each processor $12\lceil \log n \rceil$ trials. If after this many trials, it has not found accommodations for all its cells the process as a whole is aborted using the concurrent write ability.

LEMMA 4.3. The probability that the above procedure aborts is at most $1/n$.

Proof. Let Y be a random variable corresponding to the number of accommodations found after $t = 12(\lceil \log n \rceil)$ trials. Since each trial finds an accommodation with probability at least 1/2 the random variable Y is bounded from above by a binomial random variable X with $p = 1/2$ on t trials. That is $\text{Prob}(Y \leqslant x) \leqslant \text{Prob}(X \leqslant x)$ for all x.

Here we use the Chernoff bound:

$$\text{Prob}(X \leqslant \lfloor (1 - \varepsilon) pt \rfloor) \leqslant e^{-\varepsilon^2 tp/2}.$$

Setting $\varepsilon = 5/6$, $p = 1/2$, and $t = 12\lceil \log n \rceil$ we get

$$\text{Prob}(X \leqslant \lceil \log n \rceil) \leqslant e^{-(25/12)\lceil \log n \rceil} \leqslant e^{-2 \log n} \leqslant 1/n^2.$$

Thus, the probability of failure for any given processor is at most $1/n^2$. Therefore, failure probability as a whole is at most $1/n$. □

4.3. Removing a Constant Proportion of Zeros from a Random Sting

Let $\sigma = s_1 \ldots s_n$ be a random binary string where each s_i is an independent random variable that takes the value one with probability p and zero with probability $q = 1 - p$. We view σ as a sequence of live and dead cells where the ith cell is alive if $s_i = 1$ and dead if $s_i = 0$. One can remove all dead cells by computing all partial sums.

Thus, all dead cells can be removed in $O(\log n)$ time using $O(n/\log n)$ processors. We need a faster algorithm that uses only $O(\log(\log n))$ time and $O(n/\log(\log n))$ processors. But we require

only that the algorithm remove a constant proportion of the dead cells in a random string.

We shall say that an algorithm on an input string σ of length n discards k zeros if it maps the nonzero elements in a one-to-one way into a new string of length at most $n - k$.

THEOREM 4.4. There exists a ***PRAM*** algorithm ***DISCARD ZEROS*** using $O(\log(\log n))$ time and $O(n/\log(\log n))$ processors that, *for* at least $1 - 1/n$ *of* the random strings σ *of* length n discards at least $\lceil qn/2 \rceil$ zeros, $p = 1 - q$ *fixed* and n *sufficiently* large.

Proof. We partition n into intervals of size $m = \lceil c(\ln n) \rceil$ plus one last interval of size $\leq m$. We fix c as a function of q later in the proof. Each interval will be given $k = \lfloor (p + q/2)m - 1 \rfloor$ consecutive storage locations in which to store its live cells. We assign $O[m/\log(\log n)]$ processors to each interval. In $O(\log m)$ time, using the classical prefix sums algorithm, these processors place the live cells in their interval. If any interval has more live cells than storage locations then the process as a whole is aborted using concurrent write. The algorithm has thus failed on this input.

We first check that if the algorithm terminates then it has in fact it has discarded $\lceil q/2n \rceil$ zeros. The total space used for storing the ones is $\lceil n/m \rceil \lfloor (p + q/2)m - 1 \rfloor$, which is less than or equal to

$$(n/m + 1)[(p + q/2)m - 1]$$

$$= (p + q/2)n + (p + q/2)m - n/m - 1.$$

For $n \geq (p + q/2)m^2$ this sum is less than $\lfloor (p + q/2)n \rfloor$. Thus, we have discarded $\lceil q/2n \rceil$ zeros.

Before we show that the algorithm fails only on a vanishingly small fraction of the strings we analyze the number of processors and the time uses. Sincce there are $\lceil n/m \rceil$ intervals each using $O(m/\log(\log n))$ processors the total number of processors used in $O(n/\log(\log n))$. Since each interval can be packed in parallel, the total time (besides computing the parameters m and k) will just be the cost of computing all the prefix sums for a string of length m, which is $O(\log m) = O(\log(\log n))$.

The procedure fails on some interval if the number of zeros in that interval is less than $\lceil q/2m \rceil$. It will suffice to show that the probability of fewere than $\lfloor q/2m + 1 \rfloor$ zeros in an interval is small.

Note that $q/2m + 1 = (1 - \varepsilon)qm$ for $\varepsilon = 1/2 - 1/(qm)$ and for m large $\varepsilon > 0$. To analyze the probability of failure we use Chernoff bounds from Theorem 3.5. Let S_m be a binomial random variable with parameters m, q. We have the following bound on the probability of failure for some interval:

$$\text{Prob}(S_m \leqslant \lfloor (1 - \varepsilon)mq \rfloor) \leqslant e^{-\varepsilon^2 mp/2}.$$

Using our values of ε and m, and setting $c = 9/q$ we get

$$e^{-[1/2 - 1/(qm)]mq/2} = e^{-(mq/2 - 1/2)} = e^{-2\ln n - (\ln n - 1/2)} \leqslant 1/n^2.$$

The last inequality follows for $n \geqslant 2$. Now the probability of failure on any interval is upper bounded by $(n/m)1/n^2 = 1/mn$. Since $m \geqslant 2$ we get that failure occurs less than $1/n$ of the time. □

THEOREM **4.5.** ***There exists a PRAM algorithm using*** $O(\log(\log n))$ ***time and*** $O(n/\log(\log n))$ ***processors that for at least*** $1 - 1/n$ ***of the strings with b zeros discards at least*** $\min\{b/2, n/3\}$ ***zeros.***

Proof. To prove the theorem we use the algorithm from the proof of the previous theorem with $p = (n - b)/n$. The analysis of failure for the previous theorem reduces to Chernoff bounds for tails of a binomial random variable with parameters m, p. In this case the random variable is hypergeometric with parameters n, m, $n - b$. Hoeffding [H], Theorem **4,** has shown that the moments of a hypergeometric are always bound by the moments of a binomial with the same expected value. Thus, the Chernoff bounds in Theorem 3.5 can be applied directly to hypergeometric distributions. Thus the arguments used in the proof of Theorem **4.4** apply directly to this case giving an error bound of $1/n$. □

4.4. Randomized Tree Evaluation Using $O(n/\log n)$ Processors

We are now ready to describe our optimal randomized tree evaluation algorithm. The procedure is presented in Figure 5. Routine *(a)* generates for each i an upper bound x_i on the size of the work space at the ith stage of routine (c). The routine *(b)* generates in parallel all the permutations **that** will be needed in routine *(c).* We generate all the permutations at once to ensure $0(\log n)$ time.

Figure 5. An optimal randomized tree evaluation algorithm.

Procedure Randomized Tree Evaluation:
Set $x_1 \leftarrow n; \alpha \leftarrow 31/32; k \leftarrow 1; i \leftarrow 1; T_1 \leftarrow T$;
While $x_i \geq n/\log n$ **do** (a)
1) $x_{i+1} \leftarrow \lceil \alpha x_i \rceil$;
2) $i \leftarrow i + 1$
In Parallel Generate random permutations $\pi_1, \ldots, \pi_i$ (b)
of sizes $x_1, \ldots, x_i$, respectively
While k $< i$ **do** (c)
1) $T_{k+1} \leftarrow$ RANDOMIZED CONTRACT(T_k), using p processors;
2) Permute the pointers of T_{k+1} using π_{k+1};
3) Apply DISCARD **ZEROS** to the list of pointers T_{k+1}
returning at most x_{k+1} pointers;
4) $k \leftarrow k + 1$.
While $|T| > 1$ **do** (d)
$T \leftarrow$ Randomized Contraction(T)
(using a distinct processor at each vertex)

Routine (c) step (1) for each k contracts T_k to T_{k+1} generating at least $x_k/16$ dead pointers. After randomly permuting the pointers, step (2), step (3) discards at least $x_k/32$ dead pointers. When routine *(d)* is implemented, T will be stored in an array of pointers of size at most $0(n/\log n)$. Since no routine will be implemented more than $0(\log n)$ times we need only make sure that the probability of aborting at each step is $\leqslant 1/cn \log n$ for some constant c. These bounds follow from the preceding theorems and the fact that the error can be decreased to $1/n^2$ by simply running an algorithm twice.

We discuss each of the four routines: ***(a), (b),*** (c), and ***(d).*** Routine *(a)* is deterministic and thus always works. Routine *(b)* generates all $O(\log n)$ permutations needed by the algorithm. The important fact to note is that the sum of their sizes is $O(n)$. Thus, we can generate each π_i in logn time using at most $O(x_i/\log n)$ processors (see Theorem **4.2)** all with probability of failure at most $1/n$. Therefore routine *(b)* uses $O(\log n)$ time and $n/\log n$ processors. The analysis of routine (c) is slightly more complicated. By Theorem **3.7** RANDOMIZED CONTRACT will fail with probability at most $1/n$ for sufficiently large n. By Theorem **4.5** DISCARD ZEROS will fail with probability at most $1/n$ also. Step (1) will take $O(x_k/n \log n)$ time using $n/\log n$ processors. Since x_k/n is approximately α^k the time taken by (1) is geometrically decreasing in k. Therefore the total time used by (1) over all values of k is $O(\log n)$. This same analysis also applies to (2). Using

Theorem 4.5, the procedure **DISCARD ZEROS** can be speeded only by using more processors to a minimum time $O(\log(\log n))$. Thus, we must check that $x_i/n \log n \geqslant \log(\log n)$. But $x_i \geqslant 31/32n$. Thus, for n large step (3) also uses total time at most $O(\log n)$. Finally routine (d) will complete in $O(\log n)$ time by Theorem 3.6 with failure probability at most $1/n$. Thus, it follows that RANDOMIZED TREE EVALUATION will fail with probability at most $1/n$.

Using the parallel tree expansion ideas in Theorem 3.2 we get:

THEOREM 4.6. There exists a 0-sided randomized algorithm *that marks all vertices of a tree in* $O(\log n)$ time *using* $O(n/\log n)$ processors.

For deterministic dynamic tree expansion, we had enough processors so that all the trees $T_1, \ldots, T_k$ computed during dynamic tree contraction can be stored on the processors local memory using a pushdown store. Here we have fewer processors so we shall simply store the tree in common memory with back pointers from vertices in tree T_i to corresponding vertices in T_{i+1}. Since the size of the trees is decreasing geometrically the total storage is at most linear.

5. APPLICATIONS OF DYNAMIC TREE CONTRACTION

5.1. Arithmetic Expression Evaluation

Let T be a tree with vertex set V and root r. We assume each leaf **is initially assigned a value** $C(v)$, **and each** internal vertex v, with children $u_1, \ldots, u_k$, has a label $L(v)[u_1, \ldots, u_k]$ that is assumed to be of the form $\theta(u_1, \ldots, u_k)$ where $\theta \in \{+, -, \times, \div\}$. A bottom-up approach for expression evaluation is to substitute $L(u_i)$ into $L(v)[u_1, \ldots, u_k]$ for each child u_i that is a leaf, and then delete u_i. This method however requires time $\Omega(n)$ in the worse case. The results of Brent imply we can do expression evaluation in $O(\log n)$ time if we can preprocess the expression [6]; however $\Omega(\log n)^2$ time seems to be required if the expression is to be evaluated dynamically (i.e., on line).

THEOREM 5.1. Dynamic arithmetic expression evaluation can be done in $O(\log n)$ time using $O(n)$ processors deterministically and only $O(n/\log n)$ processors using a 0-sided randomized procedure.

Proof. We shall assume that the number of arguments at a vertex is at most 2. If not we assume that in $O(\log n)$ time we can convert it into such a tree. As in Brent we shall perform only one division at the end.

The values stored or manipulated will be sums, products, and differences of the initial leaf values $C(v)$. The value returned will be a ratio of these elements. The operations $\{+, -, \times, \div\}$ will have their usual interpretations, e.g., $a/b + c/d = (ad + bc)/bd$. The other main item we need is a way to represent elements from a class of many functions that is closed under composition. Here we will use ratios of linear functions of the form, $(ax + b)/(cx + d)$. We must verify that they are closed under composition:

$$\frac{a'(au + b)/(cu + d) + b'}{c'(au + b)/(cu + d) + d'} = \frac{a''u + b''}{c''u + d''}.$$

□

5.2. Arithmetic Subexpression Evaluation

By running procedure randomized tree evaluation "backward" (Figure 5) as we did in the deterministic case (Figure 3) we get:

THEOREM 5.2. ***AN subexpressions can be computed in the time and processor bounds in Theorem 5.1.***

A special case of computing all subexpressions is the linked-list ranking problem. Here we have a linked-list and we would like to compute the position in the list of all elements (see [V]).

COROLLARY 5.3. ***General a linked-list of size n, the position of each element in the list can be computed with a 0-sided randomized algorithm in $O(\log n)$ time using $O(n/\log n)$ processors.***

6. PARALLEL TREE CONTRACTION FOR TREES OF UNBOUNDED DEGREE

Up until now we have assumed that the RAKE operation could be performed in unit time. For many applications this is not the case. As we shall see in the companion paper [MR2], the RAKE operation for certain applications may be considerably more complicated than just deletion. As an example, we may need to sort

the labels assigned to the leaves of a vertex before they can be "ranked." Here, the parallel time of raking the leaves of a vertex with k children is $O(\log k)$. If we require the algorithm in this example to finish a CONTRACT completely before it is allowed to start the next CONTRACT then it is not hard to construct examples where the total cost to reduce a tree to its root will be the cost for RAKE times the logarithm of the size. As an example of where the Rake operation is not constant time in Part **2** of this paper we consider the problem of finding canonical labels for trees; here the RAKE operation consists of sorting the labels of all siblings before the leaves are removed (see [MR2] for details). It is well-known how to sort in $O(\log^2 n)$ time. Thus, the naive analysis of this algorithm would be that it runs for $O(\log^2 n)$ time. We will improve the running time by a factor of logn below.

We modify parallel tree contraction ***so*** that for those parts of the tree where CONTRACT has already finished we implement a new round of CONTRACT, i.e., CONTRACT is run asynchronously. We shall assume that the time used to remove the leaves **of** a given vertex is only a function of the number of leaves at that vertex. We should point out that the synchronous and asynchronous versions of CONTRACT may return very different answers. In the case of computing canonical forms for trees by sorting leaves both the synchronous and asynchronous algorithms are correct. The asynchronous version will be faster.

Asynchronous Parallel Tree Contraction (APTC) can be described graph theoretically by viewing it as operating on trees with special leaves that we call ***phantom leaves.*** The algorithm APTC is run in stages. Initially the tree T has no phantom leaves. We apply the procedure CONTRACT to T obtaining the tree T'. If a given vertex $v \in T$ had $k \geqslant 2$ children that are leaves excluding any phantom leaves then we add a new phantom child w to $v \in T'$. Further, if the time required for APTC to delete these k children of v is t then the phantom vertex w will persist for t stages at which time it simply disappears. Note that a given vertex may have several phantom children and a vertex with a phantom child is not a leaf. The time to execute APTC is the number of stages it takes to reduce the tree to its root and all phantom leaves to disappear.

THEOREM 6.1. ***If the cost to RAKE a vertex with k children is bounded by*** $O(\log k)$ ***then asynchronous tree contraction requires only*** $O(\log n)$ ***time.***

Proof. Suppose the time to **RAKE** a vertex with k children is bounded by $c \log k$ for $k \geq 2$ and **RAKE** for a single child can be performed in unit time. We shall analyze the time used by asynchronous parallel tree contraction by assigning weights to the vertices of the tree such that at any stage of the algorithm the weight of the tree reflects the progress made so far.

A weighted tree is a tree with weights assigned to the vertices. The weight of a tree is the sum of the weights of the vertices in the tree. In this application all vertices will have weight one except phantom leaves, which may have arbitrary real weights ≤ 1. Thus, initially, the weight of the tree is the size of the tree.

We describe in more detail how weights are assigned to phantom leaves. Suppose the time required to rake the k nonphantom leaves of a vertex v is $f(k)$. There is a subtle point that is worth pointing out. Namely, if the time to rake a vertex with k leaves varies from vertex to vertex this may dramatically affect the way the tree contracts. Our analysis depends only on an upper estimate for the time to rake a vertex. We pick β, which is a function of k, such that $\beta^{f(k)-1} k = 1$ for $f(k) > 0$. Hence $\beta < 1$ for all $k \geq 2$. The constant β will be the rate at which the phantom leaf decays. We set the weight of the new phantom leaf w of v to βk. After each successive stage we will decrease the weight on w by a factor of β until the weight equals one. In the next stage we will simply delete the phantom leaf w. Thus, the phantom leaf w will exist for $f(k)$ stages at which time it will be deleted. Note that the weight of a phantom vertex is always ≥ 1.

As in the proof of Theorem **2.1** we partition the vertices of T into two sets, ***Ra*** and ***Corn.*** We claim that the weight of ***Corn*** decreases by a factor of 1/2 at each stage while the weight of ***Ra*** decreases by a factor of at least $(4 + \beta)/5$ at each stage, where $\beta = \max\{\beta(k) | 1 \leq k \leq n\}$. Note that different phantom leaves decay at different rates. We have picked β to be the slowest such rate. The fact that *Corn* decreases by **1/2** follows by noting that the vertices in *Corn* are processed the same as in CONTRACT and their weights are all one. We next consider the case of ***Ra,*** $Ra = V_0 \cup V_2 \cup C_0 \cup C_2 \cup GC$, where V_0 is the set of leaves and phantom leaves. Since the weight on any vertex in V_0 is at least one and the weight of any vertex not in V_0 is 1 we see that that weight of V_0 is at least 1/5 of the weight of ***Ra.*** On the other hand the weight of V_0 decreases by at least β at each stage. Thus, the weight of ***Ra*** decreases by at least a factor of $4/5 + \beta/5$ at each stage.

This shows that the number of stages is bounded by logn base $5/(4+\beta)$. For a particular case of interest when $f(k) \leqslant c \log k$ for some constant c and $k \geqslant 2$ we see that β is bounded away from 1 for all n. This proves the theorem. □

7. THE RANDOM VARIABLE MATE

Let Σ_n be the space of all binary strings of length $n+1$ for n $\geqslant$ 1. Let $MATE_n$ be a random variable defined on Σ_n where $MATE_n$ equals the number of 01 patterns in a string from Σ_n. Intuitively, 0 is a female and 1 is a male.

LEMMA 7.1. *The random variable $MATE_n$ has expected value $n/4$ and variance $(n+2)/16$.*

Proof. Let $s_0 \ldots s_n$ be a random binary string. Since the expected value of $MATE_n$ substring $s_i s_{i+1}$ is $1/4$ and there are n such substrings the expectation for $s_0 \ldots s_n$ must be $n/4$. Here we used the fact that expectations sum.

To compute the variance we consider a slightly different random variable with the same probability distribution. Let S_n be the binomial random variable on binary strings of length n with $p = 1/2$. We define a random variable X with $p = 1/2$ over the space of all zero–one strings of length $n+1$ as follows:

$$X(t_0 \ldots t_n) = \begin{cases} \lceil (S_n(t_1 \ldots t_n))/2 \rceil & \text{if } t_0 = 0 \\ \lfloor (S_n(t_1 \ldots t_n))/2 \rfloor & \text{if } t_0 = 1. \end{cases}$$

To see that X is simply a change of variables of $MATE_n$ consider the mapping from $s_0 \ldots s_n$ to $t_0 \ldots t_n$ defined by $t_0 \leftarrow s_0$ and inductively $t_i = 0$ iff $s_{i-1} = s_i$. One can see that this mapping is surjective and $X(t_0 \ldots t_n) = MATE_n(s_0 \ldots s_n)$. Thus, the expected value of X is $n/4$ and we need only compute the second moment of X, $E(X^2)$.

$$\begin{aligned} E(X^2) &= 1/2 \sum_{k=0}^{n} \{\lceil k/2 \rceil^2 \operatorname{Prob}(S_n = k) + \lfloor k/2 \rfloor^2 \operatorname{Prob}(S_n = k)\} \\ &= 1/2 \sum_{k \text{ odd}} [(k^2+1)/2] \operatorname{Prob}(S_n = k) \\ &\quad + 1/2 \sum_{k \text{ even}} (k^2/2) \operatorname{Prob}(S_n = k) \\ &= 1/4 \left[\sum_{k=0}^{n} k^2 \operatorname{Prob}(S_n = k) + \sum_{k \text{ odd}} \operatorname{Prob}(S_n = k) \right] \end{aligned}$$

The first term in the sum is just **1/4** of the second moment of S_n, which is $(n^2 + n)/4$. By a straightforward examination of Pascal's Triangle the second term equals 1/2, since the sum consists of every other term in a row of Pascal's triangle, which is equal to the sum of the row above it. Thus, $E(X^2) = (n^2 + n + 2)/16$. Therefore the $\mathrm{var}(X) = E(X^2) - E^2(X) = (n+2)/16$.

By similar arguments we get the following bound on $MATE_n$:

LEMMA **7.2.** $\forall x\, Prob(\lceil S_n/2 \rceil \leqslant x) \leqslant Prob(MATE_n \leqslant x) \leqslant Prob(\lfloor S_n/2 \rfloor \leqslant x)$.

One of the referees has noted that Lemma **7.1** follows by a rather straightforward induction based on covariances. We feel that our proof while slightly longer is instructive since it shows that the random variable $MATE_n$ is very closely related to a simple binomial random variable. We conjecture that the random variable $MATE = MATE_{n_1} + \cdots + MATE_{n_m}$ where $n = n_1 + \cdots + n_m$ has all its moments bounded by the moments of S_n for $p = 1/4$. If the conjecture is true the analysis of many of the theorems could be simplified and the constant improved. We hope that this section is of some help in settling this conjecture.

ACKNOWLEDGEMENTS

This work was supported in part by **National Science Foundation Grant DCS-85-14961 and Air Force Office of Scientific Research AFOSR-82-0326 (to G.L.M.) and office of Naval Research Contracct N00014-80-C-0647 and National Science Foundation Grant DCR-85-03251 (to J.H.R.).**

NOTES

1. **Preliminary version of this paper appeared in Miller and Reif [MR1].**

REFERENCES

[AV] D. Angluin and L. G. Valiant, Fast probabilistic algorithms for hamiltonian paths and matchings, *J. Computer System Sci.* **(18): 155–193 (1979).**

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

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

[C] H. Chernoff, A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. ***Ann. Math. Statistics*** 23, 1952.

[F] Faith E. Fich, New bounds for parallel prefix circuits, in ***Proceedings of the 15th Annual ACM Symposium on Theory of Computing,*** pp. 100–109, ACM, Boston, MA, April 1983.

[H] W. Hoeffding, Probability inequalities for sums of bounded random variables, ***Amer. Statistical Assoc. J.*** 13–30 (March 1963).

[JK] N. J. Johnson and S. Katz, ***Discrete Distributions.*** Houghton-Mifflin, Boston, MA, 1969.

[LF] Richard E. Ladner and Michael J. Fisher, Parallel prefix computation, **J.** ***Assoc. Computing Mach.*** 27(4): 831–838 (October 1980).

[L] M. Loeve, ***Probability Theory,*** Vol. 1, 4th ed., Springer, Berlin, 1977.

[M] Gary L. Miller, Finding small simple cycle separators for 2-connected planar graphs, **J.** ***Computer System Sci.*** 32(3): 265–279 (June 1986).

[MR1] Gary L. Miller and John H. Reif, Parallel tree contraction and its applications. In ***26th Symposium on Foundations of Computer Science,*** pp. 478–489, IEEE, Portland, Oregon, 1985.

[MR2] Gary L. Miller and John H. Reif, Parallel tree contraction. Part 2: Further applications, ***SIAM*** **J.** ***Computing,*** submitted.

[R1] John H. Reif, On the power of probabilistic choice in synchronous parallel computations, ***SIAM*** **J.** ***Computing*** 13(1): 46–56 (1984).

[R2] John R. Reif, An optimal parallel algorithm for integer sorting, in ***26th Annual Symposium on Foundations of Computer Science,*** pp. 496–504, ACM, ***1985.***

[SV] Y. Shiloach and U. Viskin, An $O(\log n)$ parallel connectivity algorithm. **J.** ***Algorithms*** 3: 57–67 (1982).

[U] J. Uspensky, ***Introduction to Mathematical Probability,*** 1st ed., McGraw-Hill, New York, 1937.

[V] U. Vishkin, Randomized speed-ups in parallel computation, in ***Proceedings of the 16th Annual ACM Symposium on Theory of Computing,*** pp. 230–239, ACM Washington D.C., April 1984.