PARALLEL TREE CONTRACTION AND ITS APPLICATION

Gary L. Miller[1]
Department of Computer Science
University of Southern California
Los Angeles, CA 90089-0782

John H. Reif[2]
Aiken Computation Lab.
Harvard University
Cambridge, MA 02138

1. Introduction

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 "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 [5]. This "top-down" approach has several complications, one of which is finding the separators. 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 seems to add a factor of log n 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 straight forward 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 [15].

We shall use the P-RAM model of a parallel processing device see [21]. A P-RAM consists of a collection processors. Each processor is a random access machine where it can read and write in a common random access memory. In unit time they are allowed concurrent reads and concurrent writes (CRCW), as well as arithmetic operations on integers of magnitude $n^{O(1)}$. There are two natural implementations of concurrent reads. (1) if two or more processors attempt to write in a given location of common memory then one of the processor will succeed. The preformance of the algorithm should not depend on which processor succeeds. (2) In the second model concurrent reads in a given location cause detectable noise to be stored in that location. Unless otherwise stated we shall assume the first model for concurrent reads. But, most of our algorithms 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 $\leq n^r$ 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 accepts w then $w \in L$ else A is correct with probability of error $\geq 1-1/n$. Note that we have chosen $1/n$ for our error bound instead of the common value 1/2. It 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, to achieve the tighter error bound $1/n^{\alpha}$ only increases the running time by a factor of α. We say an algorithm is 0-sided randomized if it is alway correct when it terminates and the probability of termination is $\geq 1-1/n$. We often denote 0-sided and 1-sided by subscripts of 0 and 1 respectively, see [17].

All our P-RAM algorithms will only use a polynomial number of processors. We shall take considerable effort to minimize the number of processors used. Most of these results can also be expressed in terms of circuits with simultaneous depth $O(\log n)^{O(1)}$ and $n^{O(1)}$ size. We leave the discussion of circuit size to the final paper.

The Main Results of This Paper

1. We exhibit a deterministic P-RAM 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.

2. We extend the algorithms in 1. to evaluate all subexpressions using the same time and number of processors.

3. We exhibit a 0-sided randomized algorithm for testing isomorphism of trees, subtrees, and subexpressions using $O(\log n)$ time and $O(n/\log n)$ processors. We

[1]This work was supported in part by National Science Foundation grant NSFCS-80-07756 and Air Force Office of Scientific Research AFOSR-82-0326

[2]This work was supported by Office of Naval Research Contract N00014-80-C-0647 and National Science Foundation grant DCR-85-03251

also exhibit a deterministic O ($log\ n$) time algorithm using O ($n^2 log\ n$) processors for canonical forms of trees.

4. We show that the tree of 3-connected components (as defined by Hopcroft & Tarjan [9]) is constructible in O ($log\ n$) time on a P-RAM.

5. We construct an O ($log^2 n$) time P-RAM algorithm that computes explicit planar embedding of planar graphs even if the graphs are not 3-connected.

6. We construct an O ($log^3 n$) time P-RAM algorithm that computes a canonical form for planar graphs.

Previous Work

We compare each of these new results with previous work.

1. Brent [5] showed that expressions of size n could be rewritten in straight-line code of depth O ($log\ n$). Natural dynamic implementations of this work in parallel seem to require O ($log^2 n$) time.

2. Our result is a natural generalization of parallel prefix evaluation [7, 24]. Up to constant factors we use no more time or processors.

3. Ruzzo [20] shows that isomorphism of trees of degree at most log n could be done in O ($log\ n$) time. No polylog parallel algorithm was known for tree isomorphism of unbounded degree.

4. Ja'Ja' and Simon [11] give an O ($log\ n$) P-RAM algorithm for finding maximal subsets of vertices which are pairwise 3-connected, but they do not address the problem of finding the tree of 3-connected components. In particular, they do not construct embeddings of general planar graphs.

5. Ja'Ja' and Simon [11] give an O ($log^2 n$) P-RAM algorithm for constructing embeddings of 3-connected graphs but only test, in principal, if a general graph is planar.

6. No previous polylog parallel algorithm for testing isomorphism of planar graphs existed.

The body of the paper consists of 6 sections. This section states the main results of this paper and compares these new results with previous work. In section 2 we define two abstract operations on trees, RAKE and COMPRESS. We show that only O ($log\ n$) simultaneous applications variance of these operations are needed to reduce a tree to a point. In section 3 we show how to implement these operations on a randomized P-RAM in unit time using an optimal number of processors. We call this implementation Dynamic Tree Contraction. In sections 4, 5 and 6 we apply Dynamic Tree Contraction to expression evaluation, and tree isomorphism, and canonical forms for trees and planar graphs.

2. The RAKE and COMPRESS Operations

Let $T=(V,E)$ be a rooted tree with n nodes 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 node.

Let RAKE be the operation of removing 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 node. We can circumvent this problem by adding one more operation.

We say a sequence of nodes $v_1,...,v_k$ is a chain if v_{i+1} is the only child of v_i for $1 \leq i < k$, and v_k has exactly one child and that child is not a leaf. In one parallel step, we compress a chain by identifying v_i with v_{i+1} for i odd and $1 \leq i < k$. Note that if we represent T as an expression, then it is easy to find each maximal chain and its vertices in O ($log\ n$) time using O (n) processors. Let COMPRESS be the operation on T which contracts all maximal chains of T in one step. Note that maximal chains of length one are not effected by COMPRESS.

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

Theorem 1: After $\lceil log_{5/4} n \rceil$ executions 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 2 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 partition the vertices C_1 into GC_0, GC_1, and GC_2 according to whether the grandchild is in V_0, V_1, and V_2 respectively. Let $Ra=V_0 \cup V_2 \cup C_0 \cup C_2 \cup GC_0$ and $Com=V-Ra$.

To see that Ra decreases by a 1/5 after each RAKE we show that $|Ra| \leq 5|V_0|$. The inequality follows by noting that $|V_2| < |V_0|$, $|C_0| \leq |V_0|$, $|GC_0| \leq |V_0|$, and $|C_2| \leq |V_2|$.

Note that every vertex in V_1 except those of C_0 belong to a chain. Thus every vertex of Com belongs to some maximal chain. If $V_1,...,V_k$ are the vertices of a maximal chain then either $V_k \in C_2$ or $V_k \in GC_0$. In either case $V_1,...,V_{k-1}$ are the only elements in the chain belonging to Com. Thus, the number of elements in a maximal chain of Com decreases by at least a factor of 1/2 after COMPRESS. □

The type of argument used in the proof of theorem 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$(V)=Com$ 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 which we assume can be computed in parallel quickly. (Typically the vertices of the tree T will contain labels storing information relevant to the given application. The RAKE and COMPRESS operations will modify these labels, as well as the tree itself.)

As a simple example in the case when T is an expression tree over $\{\cdot,+\}$ the RAKE corresponds to the operation of 1) evaluating a node if all of its children have been evaluated or 2) partially evaluating a node if some of its children have been evaluated. The cost of applying RAKE to an expression tree is the cost of evaluating a node. If a node has been partially evaluated except for one child then the value of the node is a linear function of the child, say, $aX+b$ where X is a variable. Thus a chain is a sequence of nodes each of which is a linear function of its child. In this application, COMPRESS is simply pairwise composition of linear functions.

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

3. Dynamic Tree Contraction (Deterministic and Randomized)

3.1. Deterministic Tree Contraction

In this section we describe in more detail two implementations of COMPRESS. The first is deterministic while the second is a randomized algorithm which is given in subsection . 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 only use $O(n/\log n)$ processors and $O(\log n)$ time. In this section we assume that the trees are of bounded degree. The analysis of trees of unbounded degree is in section .

Let T be a rooted tree with node set V of size $n=|V|$ and root $r \in V$. We view each node, which is not a leaf, as a function to be computed where the children supply the arguments. For each node 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 $node(P(v))$ be the node associated with storage location $P(v)$. Figure 3-1 contains 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 $Arg(v)=Arg(node(P(v)))=1$ then
      $P(v) \leftarrow P(node(P(v)))$.
  od

Figure 3-1: A Dynamic Contraction Phase

The procedure implements the RAKE in the straight forward way; while the operation COMPRESS is implemented by pointer jumping. In line 2) of the procedure each node 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 $Arg(v)=0$ is applied 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 function of $P(v)$, composes it with its own function, and adjusts its pointer to $P(node(P(v)))$. It follows that this correctly computes the value of the expression. We next analyze the number of applications of Dynamic Contraction used.

Theorem 2: The number of applications of Dynamic Tree Contraction needed to reduce a tree of n nodes to its root is identical to 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 purpose of analysis we can discard the second chain for the analysis since it will never be evaluated. Thus a single phase of dynamic tree contraction is just CONTRACT, after discarding the unevaluatable chains. □

Note that many nodes are not evaluated, that is, for many v $Arg(v)$ is never set to 0 during any stage of Dynamic Tree Contraction. We will define a new procedure Dynamic Tree Expansion which will allow the evaluation of all nodes, i.e., each node will eventually have all its arguments after completion of the procedure. We modify Dynamic Tree Contraction so that each node keeps a push-down store $Store_v$ which is initially empty 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 $Store_V$ 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 3-2 until all nodes have all their arguments.

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

Figure 3-2: A Dynamic Expansion Phase

We must show that after successive applications of Dynamic Tree Expansion all nodes have their arguments. As in the proof of Theorem 2 we can discard those chains that have a leaf which will not be evaluated. The proof is by induction on the trees with only essential chains, as defined in the proof of the previous theorem, starting from the singleton r and finishing with the original tree T, say, $\{r\}=T_1,\ldots,T_k=T$. Now every node in T_{i+1} is either a leaf in which case we know its value or it is missing one argument which is the value of a node in T_i. In the later case this value will be supplied in one application of Dynamic Tree Expansion. This gives the following theorem.

Theorem 3: In at most $\lceil log_{5/4} n \rceil$ applications of dynamic tree contraction and $\lceil log_{5/4} n \rceil$ applications of dynamic tree expansion are needed to mark all nodes.

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 which up to constant factors uses an optimal number $O(n/log\ n)$ of processors and still runs in time $O(log\ n)$.

The analysis will follow arguments similar to those used in the proof of Theorem 1. Here we partition the vertex set V into Rake(V) and Compress(V) as defined in that proof. Again by similar arguments step 1) of RANDOMIZED CONTRACT will delete at least a 1/5 of the nodes in Rake(V). Steps 2) and 3) of randomized CONTRACT we call Randomized Pointer Jumping. The expected number of nodes of Compress(V) which are deleted in step 3c) is $m/4$ where $m=|Compress(V)|$. We cannot directly conclude that the median is also $m/4$. We can lower bound the median using the expected number and the variance of the number of nodes deleted. Since the number of deleted nodes in each maximal chain is mutually independent, the number of deleted nodes is the sum of independent random variables, one for each maximal chain. Let $C_1,\ldots,C_k$ be a list of maximal chains

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 $Arg(v)=F$ and $Arg(node(P(v)))=M$# then do
a) Push on $Store_v$ value $P(v)$;
b) $P(v) \leftarrow P(node(P(v)))$;
c) delete node($P(v)$).
od
od

Figure 3-3: A RANDOMIZED CONTRACT Phase

in T where C_i is a chain of length m_i+1 Thus, m_i of the nodes of C_i belongs to Compress(V). Let the number of deleted nodes after one application of RANDOMIZED CONTRACT be the random variable $MATE_{m_i}$. If $m=|Compress(V)|$ then the random variable which is the number of deleted nodes in one phase will be $X=MATE_{m_i}+\ldots+MATE_{m_k}$ where k is the number of maximal chains. Thus, the expected value of X is $E(X)=m/4$. By Lemma 30 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 Chebichev inequality gives the following estimate for the median of X, $\mu(X)$, see ([14] page 244).

Lemma 4: $|\mu(X)-E(X)| \leq \sqrt{2Var(X)}$

Thus $\mu(X) \geq E(X)-\sqrt{2Var(E)}$.

In our case this gives $\mu(X) \geq m/4-\sqrt{3m/8}$.
Therefore for sufficiently large m $\mu(X) \geq m/5$.

Theorem 5: For any $\epsilon>0$ and sufficiently large n RANDOMIZED CONTRACT deletes at least $(1-\epsilon)n/5$ 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 by the last lemma for m sufficiently large, say l, $m/5$ of the vertices in Compress(V) are also deleted. In the case when $m<l$ we argue as follows. For $n \geq l/\epsilon$ we have $(n-m)/5 \geq (n-l)/5 \geq (n-\epsilon n)/5 \geq (1-\epsilon)n/5$. We have shown that for n large and m small the vertices deleted by RAKE will suffice to prove the theorem. □

We next show that RANDOMIZED CONTRACT will delete at least $(1-\epsilon)n/8$ nodes with only exponentially small probability of failure for any $\epsilon>0$. Let S_n be the number of successes in n independent trials with probability p 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 from the expected value is exponentially small. This fact was observed by Uspensky [23], see [12]. These bounds are commonly known as Chernoff bounds [6]. We shall use the following simply stated bounds [3].

Theorem 6: For any $1>\epsilon>0$

$\text{Prob}[S_n \leq \lfloor(1-\epsilon)np\rfloor] \leq e^{-\epsilon^2 np/2}$ and,

$\text{Prob}[S_n \geq \lceil(1+\epsilon)np\rceil] \leq e^{-\epsilon^2 np/3}$.

We use these bounds to show:

Theorem 7: One phase of RANDOMIZED CONTRACT for any $\epsilon>0$ will delete at least $(1-\epsilon)n/8$ nodes with the probability of failure less than $e^{-c(\epsilon)n}$ where c is a positive constant only depending on ϵ.

Proof: Let n be the number of nodes in a tree T and m the number of nodes in Compress(T). If $m \leq 3n/8$ then $n-m \geq 5n/8$ nodes are in Rake(T) and therefore at least $1/5(5n/8)=n/8$ of them are deleted by RAKE. In this case $n/8$ of the nodes are deleted by RAKE alone without considering nodes deleted by COMPRESS. Thus, we may assume that $m>3n/8$. It will suffice to show that $(1-\epsilon)m/8$ of the nodes in Compress(T) are deleted by RANDOMIZED CONTRACT with small probability of failure. Let $I\subseteq$ Compress(V) be a maximum subset of nodes such that no node in I is a parent of another node in I, i.e. I is an independent set. Now each node in I is deleted independently with probability 1/4. Since the induced graph on Compress(T) is a forest, the number of nodes in $|I| \geq \lceil m/2\rceil$. Thus the number of nodes deleted is bounded below by the binomial random variable $S_{\lceil m/2\rceil}$. The probability that less than $(1-\epsilon)m/8$ nodes of Compress(T) are deleted then using Chernoff bounds is:

$$\leq Prob(S_{\lceil m/2\rceil} \leq (1-\epsilon)\lceil m/2\rceil 1/4) \qquad \leq e^{-\epsilon^2\lceil m/2\rceil/8}$$

$$\leq e^{-\epsilon^2(m/2)/16}.$$

Using the hypothesis that $m \geq 3n/8$ we get that the above probability:

$\leq e^{-\epsilon^2 3n/2^7}=e^{-c(\epsilon)n}$, where $c=\epsilon^2 3/2^7$. □

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 so that T is reduced to its root in $0(\log n)$ time using $0(n/\log n)$ processors. The important difference here is that we will be operating on an array of n nodes 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 a 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 after each phase of randomized contract with very high probability at least $1/8th$ of the processors are assign to dead pointers, Theorem 9. Thus after $0(ll\, n)$ phases, where $ll\, n=log(log\, n)$ 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 an 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 [24]:

Lemma 8: All prefix sums of a string of length n can be computed in $0(log\, n)$ time using $0(n/log\, n)$ processors.

This motivates a simple randomized tree evaluation algorithm using $0(nll\, n/log\, n)$ processors and $0(log\, n)$ time.

Procedure Randomized Tree Evaluation (Simple form)
1). Set $p \leftarrow \lceil nll\, n/log\, n\rceil$, $k \leftarrow 1$;
2). While $k \leq c(ll\, n)$ do
  $T \leftarrow$ Randomized Contraction(T) (*)
  (using p processors)
od
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)
od

Figure 4-1: A Randomized Tree Evaluation (simple form)

To see that it works in $0(log\, n)$ time we use Theorem 9. Note that for some constant c and large enough n that step 1) will reduce T to a tree on $\lceil n/log\, n\rceil$ nodes with probability of failure $\leq 1/n$. Now each execution of (*) will take $0(log\, n/ll\, n)$ time. Thus step 1) requires $0(log\, n)$ time. By lemma 8 step 2) only takes $0(log\, n)$ time. By the first remark and large enough c we have $|T| \leq n/log\, n$. Thus step 3) will only take $(log\, n)$ time with probability of failure $\leq 1/n$.

Thus the simple form of randomized tree evaluation reduces the processor count to $0(nll\, n/log\, n)$, by only "load balancing" once. To remove the last $ll\, n$ factor we will load balance between each application of (*). The goal will be to partially balance the load as apposed 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. An other algorithm appears in this proceedings [19]. In particular we show:

Theorem 9: There exist a randomized *P–RAM* algorithm which generates random permutations of n cells using $O(\log n)$ time, $0(n/\log n)$ 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 to $2n$ cells, which we call accommodations. Next we remove the unused cells using prefix calculations as described in the previous section. To get the original assignment of the n cells in $2n$ cells each of the $n/\log n$ processor will 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 trails, it has not found accommodations for all its cells the process as a whole aborts using the concurrent write ability.

Lemma 10: The probability that the above procedure aborts is at most $1/n$

Proof: Let Y be a random variable equal 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 above by a binomial random variable X with $p=1/2$ on t trials.

Here we use the Chernoff bound:

$Prob(X \leq \lfloor(1-\epsilon)pt\rfloor) \leq e^{-\epsilon^2 tp/2}$

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

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

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

4.3. Removing a Constant Proportion of Zeros From a Random String

Let $\sigma=s_1 \ldots s_n$ be a random binary string where each s_i is an independent random variable which 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 $i\,th$ 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 $0(\log n)$ time using $0(n/\log n)$ processors. We need a faster algorithm that uses only $0(ll\, n)$ time and $0(n/ll\, n)$ processors. But we only require that the algorithm remove a constant proportion of the dead cells in a random string.

We shall say that an algorithm on a input string σ discards k zeros if it reorders all but at least k zero elements of σ into a contiguous string.

Theorem 11: There exist a *P–RAM* algorithm DISCARD ZEROS using $0(ll\, n)$ time and $0(n/ll\, n)$ processors which, for at least $1-1/n$ of the random strings σ of length n, discards at least $qn/2$ zeros, p fixed.

Proof: Set $\epsilon=q/2p$ and $c=24p/q^2$. We partition n into intervals of size $m=\lceil c(ln\, n)\rceil$ plus one last interval of size $\leq m$. Each interval will be given $k=\lceil(p+q/2)m\rceil=\lceil(1+\epsilon)mp\rceil$ consecutive storage locations in which to store its live cells. We assign $0(m/ll\, m)$ processors to each interval. Using $0(\log n)$ time these processors place the live cells in its 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.

Before we show that the algorithm only fails on a vanishingly small fraction of the strings we analyze the number of processors and the time used. Since there are $\lceil n/m\rceil$ intervals each using $0(m/ll\, m)$ processors the total number of processors used is $O(n/ll\, n)$. Since each interval can be packed in parallel the total time (besides computing the parameters m and k) will just be the cost of all prefix sums for a string of length m, which is $0(\log m)=0(ll\, n)$.

To analyze the probability of failure we use Chernoff bounds Lemma 6. Let X be a binomial random variable with parameters m,p. We have the following inequality:

$Prob(X \geq \lceil(1+\epsilon)mp\rceil) \leq e^{-\epsilon^2 mp/3}$

This is an estimate that we failed on some fixed interval. Using our values of ϵ and m we get:

$Prob(X \geq k) \leq 1/n^2$

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

Theorem 12: There exist a P-RAM algorithm using $0(ll\, n)$ time and $0(n/ll\, n)$ processors which for at least $1-1/n$ of the strings with b zeros discards at least $b/2$ 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 [8] has shown that the tails of a hypergeometric are always bound by a binomial with the same expected value. Thus Chernoff bounds can be applied directly in 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 4-2. 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 insure $O(\log n)$ time. 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 1/32 of the dead pointers. When routine (d) is implemented, T will be stored in an array of pointers of size at most $O(n/\log n)$. Since no step will be implemented more than $O(\log n)$ times we need only make sure that the probability of aborting at each step is $\leq 1/cn\log n$ 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.

Using the expansion ideas in theorem 3 we get:

Theorem 13: There exists 0-sided randomized algorithm which marks all nodes of a tree in $O(\log n)$ time using $O(n/\log n)$ processors.

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 α_1 thru α_i of size x_1 thru x_i (b)
While $k<i$ do (c)
  1) $T_{k+1} \leftarrow$ RandomizedContraction(T_k), using p processors.
  2) Permute the pointers of T_{k+1} using α_{k+1}.
  3) Apply DISCARD ZEROS to the list of pointers T_{k+1} returning at most x_{k+1} pointers.
  4) $k \leftarrow k+1$.
od
While $|T|>1$ do (d)
  $T \leftarrow$ RandomizedContraction(T) using a distinct processor at each node.
od

Figure 4-2: An Optimal Randomized Tree Evaluation Algorithm

5. Applications of Dynamic Tree Contraction: Expression Evaluation

Let T be a tree with node set V and root r. We assume each leaf is initially assigned a value $C(v)$, and each internal node v, with children $u_1,\ldots,u_k$, has a label $L(v)[u_1,\ldots,u_k]$ which 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 which 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 [5]; however $\Omega(\log n)^2$ time seems to be required if the expression is to be evaluated dynamically (i.e., on line).

Theorem 14: Dynamic 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 node 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 only perform one division at the end.

The values stored or manipulated will be sums, products, and differences of the initial 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 main other item we need is a way to represent elements from a class of many functions which are 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:

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

By running procedure Randomized Tree Evaluation Figure 4-2 we get:

Theorem 15: All subexpressions can be computed in the time and processor bounds in Theorem 14.

6. Isomorphism and Canonical Labels For Trees

Let T,T' be two rooted trees with roots r and r'. We say T is isomorphic to T' if there exists a surjective map from $V(T)$ to $V(T')$ which preserves the parent relation. On the other hand Canonical Label is a map L from trees to strings such that T is isomorphic to T' iff $L(T)=L(T')$. Canonical Labels For All Subtrees of a tree T is a map L from $V(T)$ to finite strings such that for all $x,x' \in T$ the subtree rooted at x isomorphic to the subtree rooted at x' iff $L(x)=L(x')$.

Canonical labels for all subtrees can be used for code optimization. Here, one merges all nodes with common labels producing an acyclic digraph. This process is called common subexpression elimination. We first present a randomized algorithm for tree isomorphism. The height $h(v)$ of a node v in a tree T is the maximum distance from v to any of its leaves. That is, $h(v)=0$ if v is a leaf and if v has children $v_1,\ldots,v_k$ then $h(v)=1+max\{h(v_i)|1 \leq i \leq k\}$. It is a straight forward

exercise to see that the height of all nodes in a tree can be computed in time $O(\log n)$ using $O(n)$ processors deterministically and $O(n/\log n)$ processors by the RANDOMIZED CONTRACT techniques from the first part of the paper.

We canonically associate a multivariate polynomial $L(v)$ with each vertex v of the tree T. Let $x_1,x_2,\ldots$ be distinct independent variables. For each leaf v set $L(v)=x_1$. For each internal node v of height h with children $v_1,\ldots,v_k$ set $L(v)=\prod_{i=1}^{k}(x_h-L(v_i))$ using induction on the height h. Thus $L(r)$ of the root r is a polynomial $Q_T(x_1,\ldots,x_h)$ of degree $\leq n$. We may view Q_T as a polynomial over a field F. Using the fact that polynomial factorization is unique over F. We get:

Lemma 16: The subtrees rooted at v,v' are isomorphic iff $L(v)=L(v')$ over F.

To test if a polynomial $Q(x_1,\ldots,x_h)$ of degree $\leq n$ is identically zero we use an old idea which goes back to at least Edmonds. We simply evaluates the polynomial at a point and check to see if the value is nonzero. We need the following technical lemma.

Lemma 17: If A is a finite set such that $|A| \geq n^{\alpha}h$, where $\alpha \geq 1$, and $\vec{a}$ is a random element of A^h, and Q is not identically zero over F, then $\text{Prob}[Q(\vec{a})=0)] \leq 1/n^{\alpha}$

Proof: By induction it is not hard to show [10] that $\text{Prob}[Q(\vec{a} \neq 0)] \geq (|A|-n)^h/|A|^h$. Substituting $|A| \geq n^{\alpha}h$ we get $Prob[Q(\vec{a}) \neq 0] \geq (1-1/n^{\alpha}h)^h$. Thus, $\text{Prob}[Q(\vec{a})=0] \leq 1/n^{\alpha}$.

We describe the tree isomorphism algorithm in procedure form, see Figure 6-1

The most natural way to analyze the procedure Randomized$_1$ Tree Isomorphism is to assume that step 1) is performed once each time the input size doubles. In which case we may assume that the fields are given. On the other hand, is easy to see how to find finite fields of order $n^{O(1)}$ in $(\log n)^{O(1)}$ time. We shall ignore the cost here.

Procedure Randomized$_1$ Tree Isomorphism (1-sided).

1. Generate a finite field F of order $\geq hn^{\alpha}$.
2. For each node v of T or T' assign the polynomial $L(v)$ to v as above.
3. Assign each x_i a random value in F.
4. Evaluate Q_T and $Q_{T'}$ using one of the dynamic expression evaluation algorithms and return w and w'.
5. If $w \neq w'$ then output "not isomorphic" else output "probably isomorphic".

Figure 6-1: A 1-sided Randomized Tree Isomorphism Test

Theorem 18: Randomized$_1$ Tree Isomorphism tests tree nonisomorphism in $O(\log n)$ time using $O(n/\log n)$ processors with probability of being incorrect $\leq 1/n^{\alpha}$, for any fixed $\alpha \geq 1$.

We modify the algorithm into a 0-sided randomized algorithm: one that never makes an error. This algorithm will also find canonical labels for all the subtrees of the input trees T and T'. Here we will use the fact that T is isomorphic to T' iff there exists a map $L:V \cup V' \rightarrow$ Labels such that:

1. $L(r)=L(r')$
2. If v,v' are leaves then $L(v)=L(v')$
3. If v has children $v_1,\ldots,v_k$ and v' has children $v'_1,\ldots,v'_k$ and $\{L(v_1),\ldots,L(v_k)\}=\{L(v'_1),\ldots,L(v'_k)\}$ then $L(v)=L(v')$.

We use procedure Randomized$_1$ Tree Isomorphism to get a map possibly satisfying conditions 1), 2), and 3). Condition 1) is easy to check while condition 2) is always satisfied. To check condition 3) we first sort the pairs $<L(v),L(w)>$ and the pairs $<L(v'),L(w')>$ where $w(w')$ is a child of $v(v')$, respectively, in $V \cup V'$. We now simply check that the list are identical. Thus, the problem can be reduced to the cost of one sort. Both randomized and deterministic algorithms using $O(\log n)$ time and $O(n)$ processors are known for sorting [1, 13, 16]. In this proceedings the second author gives a randomized sorting algorithm using only $O(\log n)$ time with $O(n/\log n)$ processors for numbers of size $O(n^1)$ [19]. Using this result we get:

Theorem 19: Tree isomorphism and common subexpression elimination can be done with a 0-sided randomized algorithm in $O(\log n)$ time and $O(n/\log n)$ processors.

Note that this randomized procedure does not produce canonical forms for trees. We next show that canonical forms can be obtained by using sorting. The idea is to assign canonical labels to the nodes inductively by height. The leaves are labeled with zero. Suppose inductively that the children $v_1,\ldots v_k$ of v have labels $L(v_1),\ldots,L(v_k)$ then the label of v will be the concatenation of the sorted list of labels $L(v_1),\ldots,L(v_k)$ in braces. This definition of the label for T seems hard to implement in parallel since a label which takes a long time to compute may have a small lexigraphic value. We solve this problem by first sorting on the time that it takes to compute the label and then sort on the label itself. It will suffice to begin sorting when all but one child has its label and this final child's label will be placed at the end of the list. A node, which at an intermediate point of the algorithm, has one child may have a label with one free variable. The intended value of the variable is the label of the child. Thus, if the child also has only one child and its label has been computed up to a free variable we may compose the labels.

Since the labels may be as large as $O(n)$ long, it is unreasonable that two labels can be compared by one

processor in unit time. We will use the following easily proved fact.

Lemma 20: Two strings of length n can be compared in $0(1)$ time using $0(n \log n)$ processors.

Using the lemma we get:

Theorem 21: Canonical labelings for trees can be computed in $0(\log n)$ time using $0(n^2 \log n)$ processors.

To prove the theorem we must see that dynamic tree contraction only takes $0(\log n)$ time even when the tree has unbounded indegree and the cost of RAKE for a node with k children is $0(\log k)$. Here we may assume that the time to RAKE a node is independent of the size of its label and only dependent on the number of children.

Theorem 22: If the cost to RAKE a node with k children is bounded by $c \log k$ for some constant c then Dynamic Tree Contraction requires only $0(\log n)$ time.

7. Computing the 3-Connected Components

The 2-connected components of a graph are defined by an equivalence relation on the edges; two edges are equivalence if there exists a simple cycle containing both edges. The induced graphs formed from the equivalence classes of this relation are called the 2-connected components. Recently, Tarjan and Vishkin have shown how to construct the 2-connected components of a graph in O ($\log n$) time and linear number of processors on a P-RAM [22]. These components form a tree where a pair of components are adjacent if they share a vertex. The definition of the 3-connected components are more difficult to define and seem to require a more sophisticated algorithm.

Hopcroft and Tarjan give a precise algorithmic definition of the 3-connected components and show how any graph can be decomposed uniquely into a tree of 3-connected components [9]. They also give a linear time algorithm for finding the tree of 3-connected components [9]. Unfortunately, it is a highly sequential algorithm. A related question is finding the maximal subsets of vertices of size ≥ 2 which are pairwise 3-connected. We shall call these subsets the 3-sets of G. Ja'Ja' and Simon give an algorithm using O ($\log n$) time and $n^{O(1)}$ processors for finding these 3-sets [11]. There is a unique 3-connected graph associated with each 3-set. The proof and construction can be obtained by the following simple lemma.

First we define the notion of a bridge. Let $C \subseteq V$. Two edges e and e' of G are *C-equivalent* if there exists a path from e to e' avoiding C. The induced graphs on the equivalence classes of the C-equivalent edges are called the *bridges* of C. A bridge is trivial if it consists of a single edge. A pair of vertices is a separating pair if they have 3 or more bridges or 2 or more nontrivial bridges.

Lemma 23: If $C \subseteq V$ is a 3-set of G then each bridge of C contains at most 2 vertices in C. If G is 2-connected then the bridge contains exactly 2 vertices of C.

Proof: Suppose that some bridge B of C contains three vertices x_1, x_2, x_3 in C. Let p be a simple path from x_1 to x_3 in B. Let p_2 be a simple path from x_2 to a single vertex, say y of p such that $p_2 - y$ is disjoint from p. Let p_1, p_3 be the disjoint simple subpaths of p from y to x_1, x_3, respectively. Then p_1, p_2, p_3 are disjoint paths from y to distinct vertices x_1, x_2, x_3 of C. It follows that y is 3-connected to all the elements of C. This contradicts the assumption that C is a (maximal) 3-set. □

The algorithm will consists of two phases, in the first phase we shall remove all 3-sets of size ≥ 3 (proper 3-sets). This will decompose the G into a collection of disconnected subgraphs. Each subgraph will correspond to a maximal subtree of the tree of 3-connected components that contains no proper 3-sets. The second phase decomposes a 2-connected graph, which does not contain any proper 3-sets, into a tree of simple cycles and m-bonds. (An m-bond is a graph on two vertices with m edges between the two vertices.) We start with a discussion of the first phase.

Let C be a proper 3-set in G. We define two graphs $\overline{C}$ and H from C and G. Let $\overline{C} = (C, \overline{E})$ where the edge set $\overline{E}$ consist of 1) all edges in G whose end points are in C but these end points do not form a separating pair for G plus 2) a new virtual edge for each separating pair contained in C. While the graph $H = (V', E')$, where V' consists of all vertices of G minus those vertices of C that do not belong to some separating pair. The edges E' of H will consist of all the edges of G not in $\overline{C}$ plus a new virtual edge for each separating pair contained in C. The graphs $\overline{C}$ and H are constructible in O ($\log n$) time when C and the separating pairs are given. It is not hard to see that if $C_1, \ldots, C_k$ are the 3-sets we can simultaneously construct $\overline{C}_1, \ldots, \overline{C}_k$ and the graph H. If some connected component of H consists of an edge with exactly two virtual edges e and e' we shall delete the edge from H and associate e in some $\overline{C}_i$ with e' in some $\overline{C}_j$. We state a lemma about $\overline{C}$ and H.

Lemma 24: The proper 3-sets of H are precisely the proper 3-sets of G minus C. The resulting graph H, after removing all the proper 3-sets from G, will have no proper 3-sets and each connected component will be 2-connected.

We next show how to decompose a graph H into its tree of 3-connected components when H is 2-connected and has no proper 3-sets. Here, we shall use the ideas from the parallel tree contraction. Namely, 1) find all the leaves, remove them and 2) find and contract maximal chains.

Let $\{x, y\}$ be a 3-set. Then the bridges of $\{x, y\}$ are of three types 1) a simple edge, 2) a path of length two or

more and 3) a bridge containing a vertex from some other 3-set. We claim that the leaves of a tree of the 3-connected components are of 2 types: 1) a bridge of a 3-set $\{x,y\}$ consisting of a path p of length ≥ 2 plus a virtual edge from x to y. 2) A 3-set $\{x,y\}$ which contains at most one bridge that is not an edge, plus edges consisting of (a) the simple edge bridge between x and y and (b) a virtual edge for the nonedge bridge.

These leaves are constructible in parallel and each requires at most $O(\log n)$ time to construct using a P-RAM. We next characterize those 3-connected components which are simple cycles of the graph but which are vertices of the tree of 3-connected components and have valence 2. Find all pairs of paths, p_1 and p_2, and pairs of 3-sets, $\{x,y\}$ and $\{w,z\}$, satisfying the following condition: p_1 is a simple path from x to w visiting no other 3-sets and p_2 is a simple path from z to y visiting no other 3-sets. By adding a virtual edge from w to z and a virtual edge from y to x we get a simple cycle that is a valence 2 vertex in the tree of 3-connected components. It follows that we can remove all such simple cycles from H in parallel.

Thus in $O(\log n)$ time we can decompose A into a tree of m-bonds and simple cycles. We state this as a theorem.

Theorem 25: The tree of 3-connected components is constructible in $O(\log n)$ time using $n^{O(1)}$ processors.

Note that we have only described the decomposition in the case when the graph is 2-connected. It is not hard to extend this to the case of all connected graphs. In this case, the virtual objects will be both edges and vertices.

Ja'Ja' and Simon only test whether in principle a graph is planar but they do not actually construct the cyclic ordering of the darts except if the graph is 3-connected [11].

Since we now can construct the tree of 3-connected components it is not hard to see how to actual construct the embedding in general by viewing this as a tree contraction problem.

Theorem 26: Planar embedding for planar graphs are constructible in $O(\log^2 n)$ time using $n^{O(1)}$ processors.

7.1. Canonical Forms of Oriented Graphs

Let $G=(V,E)$ be an undirected graph. We associate with each edge $e=\{x,y\}$ two *darts* (x,y) and (y,x). The vertex x is the tail and y is the head of the dart (x,y). The graph G is oriented by fixing a permutation ϕ of the darts which sends tails to tails and cyclically permutates darts with the same tail. Let R be the permutation of the darts sending (x,y) to its reflection (y,x). A planar embedding of G can be specified by an orientation of G.

Witney showed that every 3-connected planar graph has exactly two planar embeddings, an embedding ϕ and its reflection ϕ^{-1} [25]. Ja'Ja' and Simon have shown that a planar embedding can be constructed using $O(\log^2 n)$ time on a P-RAM for 3-connected planar graphs [11]. Any isomorphism of a planar 3-connected graph must preserve its planar orientation up to reflection. More formally, two oriented graphs (G,ϕ) and (G',ϕ) are isomorphic if there exists a bijective map f from the darts of G to the darts of G' which preserves both adjacency and orientation, $R'f=fR$ and $\phi'f=f\phi$. Using Witney's theorem two 3-connected planar graphs G' and G are isomorphic if and only if (G',ϕ') is isomorphic to (G,ϕ) or (G,ϕ^{-1}).

Note that an isomorphism of one embedded graph onto another is determined by the image of a single dart. Given a sequence of numbers $u=(u_1,\ldots,u_k)$ and a dart e we get a unique path $e=e_0,\ldots,e_k$ where $e_i=\phi^{u_i}R(e_{i-1})$ for $1 \leq i \leq k$. Given a path of darts we can construct a unique sequence of integers by choosing the minimum $u_i \geq 0$ such that $e_i=\phi^{u_i}R(e_{i-1})$. We next show how to compute canonical sequences. These sequences will be used for canonical forms for embedded graphs.

Theorem 27: Canonical numbering for oriented graphs is computable in $O(\log n)$ time using $n^{O(1)}$ processors.

We will construct a canonical form $M(e)$ for each dart e in (G,ϕ). We then simply pick the lexically least such form. For each dart $e' \neq e$ we find the lexigraphically least number sequence over shortest paths from e to e'. Suppose the graph G has d darts. Consider a $d \times d$ matrix where each entry is a number sequence or blank. Here the basic scalar operations will be lexigraphical minimum and concatenation as opposed to $+$ and $\times$. Initially start with the matrix with all paths of length two by storing a sequence of numbers of length one. If we only restrict the number of processors to a polynomial in n then a matrix product over minimum and concatenation can be computed in $O(1)$ time. By computing $O(\log n)$ iterated powers of this matrix we get the lexigraphically minimal of all shortest paths between all pairs of vertices. Thus we get a canonical matrix $M(e)$ for each dart e in (G,ϕ). The minimum canonical matrix $M(e)$ (under lexigraphical order) will be a canonical form for the embedded graph (G,ϕ).

Note that there is an isomorphism if and only if the matrices $M(e)$, as described above, are equal. By also constructing the adjacency matrices for the reflection (G,ϕ^{-1}) and computing the minimum over the larger set of matrices we have constructed canonical forms for embedded graphs up to reflections. Using the additional fact that one can compute a planar embedding for a 3-connected graph in $O(\log^2 n)$ time on $n^{O(1)}$ P-RAM processors we get from above the following theorem:

Theorem 28: Canonical numbering of 3-connected planar graphs can be done in $O(\log^2 n)$ time using $n^{O(1)}$ P-RAM processors.

Remark: This result can be improved. By the use of the random walk techniques of Aleliunas, Karp, Lipton, Lovasz, Rackoff, and Reif [2, 18] we can decrease the number of processors by a factor of n.

7.2. Reducing the Problem of Finding Canonical Forms of Planar Graphs to the 3-Connected Case

In this section we give an O ($log^2 n$) time reduction from finding canonical forms for general graphs to that of canonical forms for 3-connected graphs. Since we have given O ($log^2 n$) time algorithms for finding canonical forms for 3-connected planar graphs this reduction implies an O ($log^3 n$) algorithm for canonical forms for all planar graphs. We state this as a Theorem.

Theorem 29: Computing canonical forms for general graphs is O ($log\ n$) time reducible to computing canonical forms for its 3-connected components.

By computing canonical forms we mean an oracle that accepts as input a 3-connected graph with labels on its darts and vertices and returns an incidence matrix unique up to isomorphism. We shall also assume that we have a list of new labels that we can add to the darts or vertices.

By the methods of the last section we can find up to isomorphism a unique decomposition of a graph into a tree of 3-connected components, where a 3-connected component is either a 3-connected graph, a simple cycle, a multibond, or a vertex. Two components are related by either identifying a virtual edge with orientation, a dart, in one with a virtual edge with orientation in the other or by identifying a virtual vertex in one with a vertual vertex in the other. We shall formally only handle the case when the identifications are edges, i.e., the graph is 2-connected. The general case is a straightforward generalization.

In O ($log\ n$) time we can find either a 3-connected component or an identified edge which is of maximum height in the tree. If the center is an edge we simply introduce a 2-bond as a new component which will be the center of the tree. Thus, we may assume that the tree is rooted.

To achieve the reduction for the theorem we need only implement the two basic tree contraction operations, RAKE and COMPRESS described in Section 2. We first discuss the operation COMPRESS.

Let C be a component with one child, where d_1 and d_2 are the darts associated with the parent and e_1 and e_2 are the darts associated with the child. We ask the oracle for 4 canonical matrices by assigning a new label X to either d_1 or d_2 and a new label Y to either e_1 and e_2. We write each matrix as a string and denote it by $M_C(d_i,e_j)$ for $1 \leq i,j \leq 2$. Let C' be the child of C and suppose the child also has only one child. Further, suppose the virtual darts are e_1,e_2,f_1, and f_2. As we did for C, we labeled e_1 or e_2 with X and f_1 or f_2 with Y and ask the oracle for the canonical labels for C', denoted $M_{C'}(e_i,f_j)$. Finally, canonical labels for the pair C,C' will be:

$M(d_i,f_j)$=lexigraphical minimum of

$$\{M_C(d_i,e_k),M_{C'}(e_k,f_j)\} \text{ for } k=\{1,2\}. \qquad (*)$$

Thus the operation COMPRESS is achieved by finding the four labels for each component with an only child and combining labels using (*). If C' had no children then we return with only two labels for the pair C,C', one for d_1 and one for d_2.

The RAKE operation is much simpler, in the case when the leaf C is not an only child. If d_1 and d_2 are its virtual darts we ask for canonical forms for C, where either d_1 and d_2 is assigned the label X. These labels are then assigned to the appropriate dart of the parent of C. Using the analysis of CONTRACT given by Theorem 1 we get an O ($log\ n$) time reduction.

8. The Random Variable Mate

Let Σ be the space of all zero one strings of length $n+1$ for $n \geq 1$. Let $MATE_n$ be a random variable defined on Σ where $MATE_n$ equals the number of 01 patterns in a string from Σ.

Lemma 30: The random variable $MATE_n$ has expected value $n/4$ and variance $(n+2)/16$.

Proof: Let $s_0 \cdots s_n$ be a random strings of zeros and ones. Since the expected value of $MATE_2$ substring $s_i s_{i+1}$ is 1/4 and there are n such substrings the expectation for $s_0 \cdots s_n$ 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:

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

To see that X is simply a change of variables of $MATE$ consider the map from $s_o \cdots s_n$ to $t_o \cdots t_n$ defined by $t_o \leftarrow s_o$ and inductively $s_i=0$ iff $s_{i+1}=s_i$. One can see that this map is surjective and $X(s_o \cdots s_n)=MATE(t_o \cdots t_n)$. Thus the expected value of X is $n/4$ and we need only compute the 2nd moment of X, $E(X^2)$.

$$E(X^2)=1/2\sum_{k=0}^{n}\{\lceil k/2 \rceil^2 Prob(S_n=k)+\lfloor k/2 \rfloor^2 Prob(S_n=k)\}$$

$$=1/2\sum_{k\ odd}(k^2+1)/2 Prob(S_n=k)+1/2\sum_{k\ even}k^2/2 Prob(S_n=k)\}$$

$$=1/4(\sum_{k=o}^{n} k^2 Prob(s_n=k)+\sum_{k\ odd} Prob(S_n=k))$$

The first term in the sum is just 1/4 of the 2nd moment of S_n which is $(n^2+n)/4$. By a straight forward examination of Pascal's Triangle the second term equals 1/2. Thus, $E(X^2)=(n^2+n+2)/16$. Therefore the $var(X)=E(X^2)-E^2(X)=(n+2)/16$. □

Next consider the random variable $MATE_n$ over all zero-one strings of length $n+1$ which begin with a zero. By similar argument as above we get:

Lemma 31: The random variable MATE over the space $0\{0,1\}^n$ has expected value $(n+1)/4$ and variance $(n+1)/16$.

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

Lemma 32: $\forall x Prob(\lceil S_n/2 \rceil \leq x)$
$\leq Prob(MATE_n \leq x) \leq Prob(\lfloor S_n/2 \rfloor \leq x)$.

References

1. M. Ajtai, J. Komlos, and E. Szemeredi. An 0(nlog n) Sorting Network. Proc. 15th Annual Symposium on the Theory of Computing, , 1983, pp. 1-9.

2. R. Aleliunas, R. H. Karp, R.H. Lipton, L. Lovasz, and C. Rackoff. Random Walks, Universal Traversal Sequences, and Complexity of Maze Problems. Proc. 20Th Annual Symposium on Foundations of Computer Science, IEEE, 1979, pp. 218-223.

3. D. Angluin, and L. G. Valiant. "Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings". *J. Comp. Syst. Sci.*, 18 (1979), 155-193.

4. I. Bar-On, and U. Vishkin. "Optimal Parallel Generation of a Computation Tree Form". *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 348-357.

5. R.P. Brent. "The Parallel Evaluation of General Arithmetic Expressions". *JACM 21*, 2 (April 1974), 201-208.

6. H. Chernoff. "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations". *Annals of Math. Statistics 23* (1952).

7. E. F. Fich. New Bounds For Parallel Prefix Circuits. Proc. of The Fifteenth Annual ACM on Theory of Computing, ACM, , 1983, pp. 100-109.

8. W. Hoeffding. "On the Distribution of the Number of Successes in Independent Trials". *Ann. of Math Stat.*, 27 (1956), 713-721.

9. J. E. Hopcroft, and R. E. Tarjan. "Dividing a Graph into Triconnected Components". *SIAM Journal on Computing 2*, 3 (September 1973), 135-158.

10. O. H. Ibarra, and S. Moran. "Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs". *J. of the ACM 30*, 1 (January 1983), 217-228.

11. J. Ja'Ja', and J. Simon. "Parallel Algorithms in Graph Theory: Planarity Testing". *SIAM Journal Computer 11*, 2 (May 1982), 314-328.

12. N. J. Johnson, and S. Katz. *Discrete Distributions*. Houghton Mifflin Comp., Boston, MA, 1969.

13. T. Leighton. Tight Bounds on the Complexity of Parallel Sorting. Proc. 16th Symp. Annual ACM on Theory of Computing, ACM, Washington, D. C., April, 1984, pp. 71-80.

14. M. Loeve. *Probability Theory*. Springer, Berlin, 1977.

15. G.L. Miller. "Finding Small Simple Cycle Separators For 2-Connected Planar Graphs". *JCSS* (to appear).

16. J. H. Reif, and L. G. Valiant. A Logarithmic Time Sort for Linear Size Networks. Proc. 15th Annual ACM Symp. on the Theory of Computing, ACM, 1983, pp. 10-16.

17. J. Reif. "On the Power of Probabilistic Choice in Synchronous Parallel Computations". *SIAM J. Computing 13*, 1 (1984), 46-56.

18. J. H. Reif. "Symmetric Complementation". *JACM 31*, 2 (April 1984), 401-421.

19. J. H. Reif. An Opimal Parallel Algorithm for Integer Sorting. Proc. of 25th Annnual Symp. on Foundations of Computer Science, ACM, 1985.

20. W. L. Ruzzo. "On Uniform Circuit Complexity". *Journal of Computer and System Sciences 22*, 3 (June 1981), .

21. Y. Shiloach, and U. Viskin. "An 0(logn) Parallel Connectivity Algorithm". *J. of Algorithms 3* (1982), 57-67.

22. R.E. Tarjan, and U. Vishkin. Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time. 25th Annual Symp. on Foundations of Computer Science, IEEE, 1984, pp. 12-22.

23. J. Uspensky. *Introduction to Mathematical Probability*. McGraw-Hill, New York, 1937.

24. U. Vishkin. Randomized Speed-Ups in Parallel Computation. Proc. of the 16th Annual ACM Symp. on Theory of Computing, ACM, Washington, D.C., April, 1984, pp. 230-239.

25. H. Witney. "A Set of Topological Invariant For Graphs". *American Journal Math 55*, (1937), 321-335.