SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1128–1147, December 1991

009

PARALLEL TREE CONTRACTION PART 2: FURTHER APPLICATIONS*

GARY L. MILLER† AND JOHN H. REIF‡

Abstract. This paper applies the parallel tree contraction techniques developed in Miller and Reif's paper [Randomness and Computation, *Vol.* 5, S. Micali, ed., JAI Press, 1989, pp. 47–72] to a number of fundamental graph problems. The paper presents an $O(\log n)$ time and $n/\log n$ processor, a 0-sided randomized algorithm for testing the isomorphism of trees, and an $O(\log n)$ time, n-processor algorithm for maximal subtree isomorphism and for common subexpression elimination. An $O(\log n)$ time, n-processor algorithm for computing the canonical forms of trees and subtrees is given. An Ologn time algorithm for computing the tree of 3-connected components of a graph, an $O(\log^2 n)$ time algorithm for computing an explicit planar embedding of a planar graph, and an $O(\log^3 n)$ time algorithm for computing a canonical form for a planar graph are also given. All these latter algorithms use only $n^{O(1)}$ processors on a Parallel Random Access Machine (PRAM) model with concurrent writes and concurrent reads.

Key words. parallel algorithms, tree contraction, graph isomorphism, graph connectivity, subexpression, elimination

AMS(MOS) subject classifications. 05C05, 05C10, 05C40, 68Q25, 68R25

1. Introduction. In the previous companion paper [29], we introduced a bottom-up technique for processing a tree which we named *Parallel Tree Contraction.* This technique is in many cases preferable to previously utilized top-down techniques for processing trees which require a precomputation to find the nodes of a tree that separated the tree into pieces of size at most $\frac{2}{3}$ the tree size. Our first paper considered expression evaluation as our first example and prime application of CONTRACTION. Our main results were an $O(\log n)$ time using n processor deterministic algorithm, as well as an $O(\log n)$ time using $n/\log n$ processor randomized algorithm for tree contraction. The example and application of tree contraction given in Part I [30] were dynamic expression evaluation. Part II will give some further applications. This second paper presumes that the reader has knowledge of our companion paper. The goal of this paper is to apply CONTRACTION to a wide variety of graph problems.

We will assume throughout this paper the Parallel Random Access Machine Model (PRAM), which we also assume can perform concurrent reads and writes (see [40]).

The discussion begins in §2, where we present a zero-sided randomized algorithm which tests the isomorphism of trees in $O(\log n)$ time using $n/\log n$ processors, and which tests the isomorphism of maximal subtrees and subexprasions in $O(\log n)$ time using n processors. We also exhibited a deterministic $O(\log n)$ time algorithm which uses n log n processors for computing the canonical forms of trees. Previously, Ruzzo [33] showed that isomorphism of trees of degree at most logn could be tested in $O(\log n)$ time. No polylogarithmic parallel algorithm was previously known for isomorphism of unbounded-degree trees.

* Received by the editors August 31, 1987; accepted for publication (in revised form) December 20, 1990. The preliminary version of this paper appeared in 26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, 1985, pp. 478–489.

† School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890. The work of this author was supported in part by National Science Foundation grant CCR-8713489.

‡ Computer Science Department, Duke University, Durham, North Carolina 27706. The work of this author was supported in part by DARPA/ARO contract DAAL03-88-K-0195, Air Force contract AFOSR-87-0386, DARPA/ISTO N00014-80-C-0458 and N00014-91-J-1985, NASA subcontract 550-63 of primecontract NAS5-30428.

In §3, the tree of 3-connected components (as defined by Hopcroft and Tarjan [16]) is constructed in $O(\log n)$ time on a PRAM. Previously, Ja'Ja' and Simon [18] gave an $O(\log n)$ time PRAM algorithm for finding maximal subsets of vertices, which are pairwise 3-connected; but they did not address the problem of finding the tree of 3-connected components. In the case of 3-connected graphs, they constructed the planar embedding in $O(\log^2 n)$ time on an Exclusive Read and Exclusive Write (EREW) PRAM, but it is easy to see that their algorithm required only $O(\log n)$ time, using the Concurrent Read and Concurrent Write (CRCW) model. They did not construct embeddings of general planar graphs. In §3, an $O(\log^2 n)$ time PRAM algorithm is given that computes the explicit planar embedding of planar graphs even if the graphs are not 3-connected.

Section 4 presents an $O(\log^3 n)$ time PRAM algorithm that computes a canonical form for planar graphs. No polylogarithmic parallel algorithm for testing the isomorphism of planar graphs previously existed.

Section 5 presents an NC reduction from the problem of computing canonical forms of a general graph to the problem of canonical forms for 3-connected graphs. This is an $O(\log n)$ time reduction using $n^{O(1)}$ processors on a PRAM.

Finally, §6 references extension and further applications of the parallel tree contraction technique that have been done since the original writing of this paper.

All our PRAM algorithms use only a polynomial number of processors. Effort shall be taken to minimize the number of processors used. Most of these results can also be expressed in terms of circuits with simultaneous depth: $(\log n)^c$ and n^k size, for fixed constants c and k.

2. Isomorphism and canonical labels for trees. Let T and T' be two rooted trees with roots r and r' and vertex sets $V(T)$ and $V(T')$, respectively, where $|V(T)| = n$. T is *isomorphic* to T' if there exists a bijective map from $V(T)$ to $V(T')$ which preserves the parent relation. A map L from trees to strings such that T is isomorphic to T' if and only if $L(T) = L(T')$ is called a *canonical label.* A subtree T' of a rooted tree T is said to be an *induced subtree* if there exists a vertex v of T such that the vertices of T' are v and all the descendants of v in T. This paper considers only the induced subtrees. Thus, a subtree is assumed to be an induced subtree (note induced subtrees are also termed maximal subtrees in the literature). *Canonical labels for all induced 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 is isomorphic to the subtree rooted at x' if and only if $L(x) = L(x')$. All results to follow will apply to unrooted trees as well.

Canonical labels for all induced 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.* First, a randomized algorithm for tree isomorphism is presented.

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; otherwise, if v has children, $v_1, \cdots, v_k$, then $h(v) = 1 + \max\{h(v_i) | 1 \le i \le k\}$. It is a straightforward exercise to see that the height of all nodes in a tree with n nodes can be computed deterministically by *Parallel Tree Contraction* (PTC) in $O(\log n)$ time using n processors, or alternatively, by using $n/\log n$ processors by the randomized version of *Parallel Tree Contraction,* as discussed in the first part of this paper [29].

A multivariate polynomial Q_v is canonically associated with each vertex v of the tree T. Let $x_1, x_2, \ldots$ be distinct independent variables. For each leaf v, set $Q_v = 1$. For each internal node v of height h with children $v_1, \ldots, v_k$, set $Q_v = \prod_{i=1}^{k} (x_h - Q_{v_i})$

using induction on the height h. Thus, the polynomial Q_r of the root r with height h is a polynomial $Q_T(x_1, \ldots, x_h)$ of degree less than or equal to n. Q_r is viewed as a polynomial over a finite field F. Using the fact that polynomial factorization is unique over F, Lemma 2.1 follows.

LEMMA 2.1. *The subtrees rooted at v and v' are isomorphic if and only if $Q_v = Q_{v'}$ over a field F.*

To test if a polynomial $Q(x_1, \ldots, x_h)$ of degree less than or equal to n is identically zero, an old idea, which goes back at least as far as Edmonds, is used [34]. The polynomial is evaluated at a random point and checked to see if the value is nonzero. In this section the following technical lemma is used which is similar to a lemma in [17].

LEMMA 2.2. *If F is a finite field of size p, p prime, such that $p \geq n^{\alpha+1}h$, $\alpha \geq 1$, a' is a random element of F^h, and $Q(x_1, \ldots, x_h)$ is a polynomial of degree less than or equal to n which is not identically zero over F, then* $\text{Prob}[Q(\vec{a}) = 0] \leq 1/n^{\alpha}$.

Proof. We first show by induction on h (see [17]) that the polynomial Q has at least $(p-n)^h$ points for which it is not zero. For the case $h = 1$, Q has at most n roots out of a possible p elements. Thus, Q has at least $p - n$ nonzero points. Suppose the claim is true for all polynomials with h variables, and Q is polynomial in at most $h+1$ variables. In this case, Q can be written as a polynomial in the first variable x with coefficients being polynomials in at most h variables. At least one of the coefficients Q_1 must be a polynomial which is not identically zero. Thus, there are at least $(p-n)^h$ points for which Q_1 is not zero. Now, for each one of these points there are $p - n$ values of x in F for which Q is not zero. Therefore, Q has at least $(p-n)^{h+1}$ points for which it is not zero. Since a' is a random element of F^h, the above can be written as a probability: $\text{Prob}[Q(\vec{a} \neq 0)] \geq (p-n)^h/p^h = (1-n/p)^h$.

Substituting $n^{\alpha+1}h \leq p$ for p yields $\text{Prob}[Q(\vec{a}) \neq 0] \geq (1 - (1/n^{\alpha}h))^h$. Since $(1 - (1/n^{\alpha}h))^h \geq (1 - 1/n^{\alpha})$, the desired inequality, $\text{Prob}[Q(\vec{a}) = 0] \leq 1/n^{\alpha}$, is obtained. □

The tree isomorphism algorithm is described in procedure form (see Fig. 1). Two different procedures have actually been given, depending on whether one implements step (1) or step (1'). If step (1') is implemented, it must have access to a very small table of at most $O(\log n)$ prime integers. This table of prime integers, PT, needs to only contain *one* prime between 2^t and 2^{t+1} for each t. The existence of the primes is guaranteed by Bertrand's postulate (see [15]). As Theorem 2.3 will show, isomorphism of trees of size less than or equal to n can be tested using a table of $O(\log n)$ primes, each of value less than or equal to $n^{O(1)}$. This table can be generated in random polynomial time. To generate the table of primes, we need an estimate on the number of primes in an interval of size n to $2n$ (see [32] and a random polynomial-time primality test, [35], [24]). However, if only step (1) is used, a uniform algorithm in the usual sense is obtained. Our analysis of the uniform algorithm shows only that the probability of error is less than $\frac{1}{2}$. On the other hand, the probability of error using the table of primes is at most $1/n$. In step (4), the *Asynchronous Tree Contraction* algorithm [29] is used, since the time to RAKE a node with k children will be $O(\log k)$.

THEOREM 2.3. *Randomized Tree Isomorphism using step (1) tests tree nonisomorphism in $O(\log n)$ time using $n/\log n$ processors with the probability of error less than or equal to $1/2$. If a table of primes is given, then the procedure works with a probabilzty of error of at most $1/n^{\alpha}$.*

Proof. The case when a table PT of primes is used follows by a straightforward

Procedure Randomizedl Tree Isomorphism (One-sided)

(1) Pick a random integer m in the range $(hn^{\alpha+1})^2 \leq m \leq 2(hn^{\alpha+1})^2$.
(1') Pick a prime m in the range $hn^{\alpha+1} \leq m \leq 2hn^{\alpha+1}$ of the given list of primes PT.
(2) For each node v of T or T', assign the polynomial Q_v to v as described above.
(3) Assign to each x_i a random value between 1 and m.
(4) Evaluate Q_T and $Q_{T'}$ using one of our dynamic expression evaluation algorithms [29] and return w and w', respectively.
(5) **If** w $\neq$ w', **then** output "not isomorphic,"
else output "isomorphic."

FIG. 1. *A one-sided randomized tree isomorphism test.*

calculation using the last lemma. In this case, the algorithm tests if the polynomial $Q = Q_T - Q_{T'}$ is identically zero or not. By the last lemma, the probability that a random element is a zero of Q is at most $1/n^{\alpha}$.

Suppose a random integer is used instead of picking a prime from a table. In this case, the probability that the largest prime factor of a random integer m has size at least $\sqrt{m}$ is at least $\frac{2}{3}$ (see [20]). For Q to be zero at some point modulo m, it must be zero modulo p. Thus, at least $\frac{2}{3}$ of the time, m will have a prime factor of size at least $hn^{\alpha+1}$, in which case steps (2)–(5) will be executed with an error of at most $1/n^{\alpha}$. For a sufficiently large n the probability of error is at most $\frac{1}{2}$. □

Note that the main source of error is step (1), not steps (2)–(5). This fact is used in the next algorithm. Next, the algorithm is modified into a zero-sided randomized algorithm, i.e., one that never makes an error. The idea of the algorithm will be to modify *Procedure Randomizedl Tree Isomorphism so* that it outputs a value €or each subtree of T and T'. Assuming that these values are the correct labels for each subtree, these values are used to find an isomorphism. Note that we can easily test whether or not this map is an isomorphism. This modified procedure is called *Randomized1 Label Generation.* More precisely, steps (4) and (5) are replaced with a step that evaluates all subpolynomials.

This new algorithm will also canonically label the set of all induced subtrees of a tree. But this does not give a canonical label for trees, since there is an exponential number of trees and only a polynomial number of labels. This last problem will be addressed later on in the paper.

The problem of testing the isomorphism of trees can be reduced to the problem of canonically labeling all induced subtrees of a tree, as follows:

- Viewing the two trees as subtrees of a larger tree.
- Asking for the labeling of all its subtrees.
- Checking whether or not the labels on the two roots of the subtrees are the same.

Thus, our attention is restricted to the problem of canonically labeling all induced subtrees. The following lemma will be used here.

LEMMA 2.4. *A map L is a canonical labeling of all induced subtrees of T if and only if:*

1. *If* v,v' *are leaves, then* $L(v) = L(v')$;
2. $L(v) = L(v')$ *if and* only *if* $\{L(v_1), \cdots, L(v_k)\} = \{L(v'_1), \cdots, L(v'_k)\}$, *where* $v_1, \ldots, v_k$ *are the children of* v *and* $v'_1, \ldots, v'_k$ *are the children of* v'.

Proof. The proof is a straightforward induction on the height of subtrees. One must show that two subtrees are isomorphic if and only if they have the same labels. Condition *1* states that subtrees of height 0 (leaves) are isomorphic, while condition *2* gives us the inductive step. □

The labels generated by *Procedure Randomized1 Label Generation* clearly satisfy condition *1*. Only condition *2* remains. Note that if $\{L(v_1), \cdots, L(v_k)\} = \{L(v'_1), \cdots, L(v'_k)\}$, then, clearly, $L(v) = L(v')$. Thus, one tests only that nodes with the same label have the same set of labels on their children. One simply sorts the nonleaf vertices by their label value obtaining ordered linked lists of vertices with the same labels. It will suffice to check that consecutive vertices with the same label have children that have the same set of labels. To test this latter condition for each node, one must sort the labels of each node's children. Next, only pairs of linked lists are checked for equality. Thus, all subtrees can be canonically labeled in the cost of two sorts of less than or equal to n numbers where each number is of the size $O(\log n)$. Both randomized and deterministic algorithms using $O(\log n)$ time and n processors are known for sorting [2], [31], [8].

Using this result yields the following theorem.

THEOREM *2.5. Tree isomorphism and common subexpression elimination can be performed with a O-sided randomized algorithm in* $O(\log n)$ *time using* n *processors with an error probability of* $1/n$, *given a table of* $O(\log n)$ *primes each of value less than or equal to* $n^{O(1)}$; *otherwise, the error probability is at most* $\frac{1}{2}$.

Proof. The tree T to be labeled will have n associated polynomials, one for each subtree. *Procedure Randomizedl Label Generation* must be run with enough reliability so that any two of the n polynomials will have distinct values if their subtrees are not isomorphic. In the worst case, the difference of all pairs of polynomials must have a nonzero value. This implies that $\alpha = 3$ can be picked so that the probability of any one of the n^2 polynomials being nonzero will be at most $1/n^3$. In the case where a random integer is picked; i.e., step (1) is executed, simply note that the probability of error is at most $\frac{1}{3}$ and it comes only for the first step, not the others. Thus, the random integer case works with a probability of error of at most $\frac{1}{2}$. □

The remainder of this section exhibits a fast deterministic algorithm for canonical labelings of trees. Note that the randomized procedure developed in Theorem *2.5* does not produce canonical forms for trees. 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, $v_1, \cdots, v_k$, of v have labels $L(v_1), \cdots, L(v_k)$; then the label of v will be the concatenation of the sorted list of labels $L(v_1), \cdots, L(v_k)$, including a left and right parenthesis. By Lemma **2.4** this gives a canonical label for trees. 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 lexicographic value. This problem is solved by first sorting the children of a node based on the time in which its label was computed and then sorting the children on their label value.

The discussion begins with a simpler $O(\log^2 n)$ time parallel algorithm. Here, the children of a node are sorted when all but at most one child has its label. If this final child exists, it is placed at the end of the list. A fixed place in the list is left for the missing value. A node at an intermediate point of the algorithm which has one child may be viewed as having a label with one free variable. The intended value of the variable is the label of the child. Thus, if its child also has only one child and its label

has been computed up to a free variable, then the labels may be composed; i.e., apply COMPRESS.

Since the labels may be as large as $O(n)$ long, it is unreasonable to expect that two labels can be compared by one processor in unit time. However, two characters can certainly be compared in $\mathbf{0(1)}$ time by one processor. This implies the following well-known lemma.

LEMMA 2.6. *The comparison of two strings of length n can be performed in $O(1)$ time using n processors.*

Theorem **2.7** follows from the preceding lemma.

THEOREM 2.7. *Canonical labelings for trees can be computed in $O(\log^2 n)$ time using n processors.*

To see that the above algorithm works in $O(\log^2 n)$ time, simply note that each RAKE takes at most $O(\log n)$ time and that CONTRACT is applied at most $O(\log n)$ times by the results of [29]. The bound of n on the number of processors is obtained as follows. Initially only the leaves have their labels, and the sum of their lengths is at most n. The labels on internal nodes will be the concatenation of the leaf labels below it plus separating symbols, say, left and right parentheses. Thus, the length of the label of an internal node is linear in the number of nodes in its subtree. Since only leaves are ever sorted by the algorithm, the sum of the length of the strings sorted in any RAKE is at most $O(n)$. Thus, we need only n processors. □

Our $O(\log n)$ time algorithm is slightly more complicated. Our approach begins by sorting labels at a node as soon as they arrive. That is, we first order the children of a node based on the time each child's label arrives. Among those children whose labels arrived at the same time, we further order them by their label values. In general, this labeling returns a different canonical form and label from the previous algorithms, but it is also canonical, since the ordering of the tree is, up to isomorphism, independent of how the tree is given.

Ignoring for the moment the cost of collecting labels together so that they may be sorted in parallel, the algorithm will take $O(\log k)$ steps to remove the k leaves of a node. Thus we have an algorithm which removes the k leaves of a node in $O(\log k)$ and, therefore, by the results of [29], it will run for only $O(\log n)$ time when run asynchronously.

The labels that arrive at the same time must be coalesced so that they are "ready" to be sorted. We cannot afford to coalesce the labels after they arrive, since the cost to coalesce the labeled children may be a function of all the children of the node; thus, the overall running time may grow faster than $O(\log n)$. We circumvent the problem of coalescing the labels on-line by simply computing when the labels will arrive, without sorting, followed by a second phase where we sort these "times" offline.

Recall that each nonleaf node v has associated with it an array of storage locations, one for each child. Each storage location is used for the label of the child and will be used when its label has been computed. In the preprocessing phase, the storage locations are rearranged by sorting the children by arrival times.

As mentioned above, the time when a given value will arrive in the preprocessing phase is determined without actually computing the values. These times are then used to sort the children of each node. Let c be an integer greater than or equal to 4, such that deterministic parallel sorting of $k \geq 2$ numbers can be performed in $f(k) = c\lceil \log k \rceil + 2 + 6$ time on a Concurrent Read and Concurrent Write (CRCW) PRAM, where δ is a constant yet to be determined. Since $f(k)$ can be easily computed, the parallel sorting algorithm can be slowed down so that it takes exactly $f(k)$ time

to sort a string of length k. Let the *label-tzme* of a node in a tree T be the time at which the node gets its label when the hypothetical canonical labeling algorithm is run on T. Next, the label-time for each node is computed.

Both RAKE and COMPRESS of the above algorithm assume the labels that need to be sorted are consecutive. COMPRESS is a straightforward simulation, since each COMPRESS step takes only unit time. The simulation of RAKE is more subtle. We will now show how to determine when each node becomes either a leaf or a parent of a single child. The label-time of a leaf is 1. If a node v is at no time the parent of a single child, then the label-time of v is $\max\{f(K_i) + i\}$, where K_i is the number of children of v whose label-time is i. If, at some point, v becomes the parent of a single child, then that time will be $\max\{f(K_i) + i\}$, where the maximum is over all children except for the last child processed. Then label-time can be computed by the simulation of COMPRESS. In either case, only the value $\max\{f(K_i) + i\}$ need be computed on or before time $\max\{f(K_i) + i\}$. The value is actually computed by time $\max\{2\lceil \log K_i \rceil + i + 4\}$ (see Lemma 2.8). First, the K_i's are computed, then the $\max\{f(K_i) + i\}$ is computed from the K_i's in unit time.

By the results from [29], the largest value of any label-time will be at most $O(\log n)$. A vector of integers is initially associated with each storage location of a nonleaf node v, and all entries are zero. If the label-time for the child of a node arrives at time i, then 1 is added to position i of this vector, and the vector is marked to indicate that its time is known. A marked vector can be combined with a neighboring left or right vector, either marked or unmarked. The combination of the two vectors is simply the vector sum, and this procedure is considered a COMPRESS-like operation applied to consecutive vectors. If only one of the two vectors is marked, then the combined vector is considered unmarked; otherwise, it is considered marked. We assume that we have $O(\log n)$ processors per node.

We shall implement the above compress-like operation using a variant of Wyllie's algorithm for list-ranking [40]. We consider our list of vectors as a linked-list. As in Wyllie's algorithm, the last element points to nil. For booking reasons, add a new pointer at the beginning of the list. The algorithm finishes when the new beginning pointer points to nil. At each stage, a node may update its pointer if it is pointing to a marked vertex that is not nil. When a node updates its pointer, it also adds the value of the parent's vector to itself. Therefore, this is a CREW algorithm.

A maximal consecutive sequence of marked vectors is called a *run.* Note that the above procedure applied to a *run* will decrease the length of the run by at least $\frac{1}{2}$. At some point, the sequence of vectors will be reduced to a single vector (the new vector added to the beginning of the list) whose ith value is K_i. In unit time, K_i is replaced with $f(K_i) + i$. Also, in unit time, the maximum of logn values can be computed using $O(\log^2 n)$ processors. We use a processor P for each pair of values. The processor P will cancel the smaller of its two values. The remaining value is the maximum. We will assume that the above two-unit time calculations are performed in at most δ machine steps.

It remains to show that the vector values K_i are computed "on time." That is, the vector of values K_i is computed by time $\max\{f(K_i) + i - \delta\} \leq \max\{4\lceil \log K_i \rceil + i + 4\}$ for each node. The problem is abstracted to the following conceptually easier problem: a list of characters, each of which is initially the letter I for inactive, is presented; i.e., the string I^n is given. At time i, a subset of K_i of the characters I change, to A. Each A is now thought of as an active character. At each time step, a run of t A's is replaced by a run of $\lfloor t/2 \rfloor$ A's. This process is called *ACTIVATE and COMPRESS.*

LEMMA 2.8. *The process ACTIVATE and COMPRESS Wall terminate in the empty string by a time of at most* $\max\{2\log K_i + i + 2\}$.

Proof. Suppose that $K_1, \cdots, K_m$ is a sequence of activations where m equals the maximum i such that $K_i \neq 0$. Further, let $l = \max\{2\log K_i + i\}$, **for** $i = 1$ to m. Note that $l \geq m \geq 1$.

Let Δ_i be the number of A's in the string at time i after the ith list activation. At time i, there are K_i A's added to the string while COMPRESS reduces the number of A's by one-half. Thus, the contribution of the K_i As at time $t \geq i$ is bounded by $K_i/2^{t-i}$. This gives the following inequality for $t \geq m$:

$$\Delta_t \leq \frac{K_1}{2^{t-1}} + \cdots + \frac{K_m}{2^{t-m}}. \tag{1}$$

Using the fact that for all i, $2\log K_i + i \leq l$ implies $K_i \leq 2^{(l-i)/2}$, we substitute this inequality into (1), yielding

$$\Delta_l \leq \frac{1}{2^{l/2}} + \cdots + \frac{1}{2^{(l-m)/2}}.$$

Since the right-hand side is a geometric series in $1/\sqrt{2}$ beginning with $1/\sqrt{2}$, it follows that $\Delta_l \leq 1/(\sqrt{2} - 1) < 3$. Since A decreases by at least $\frac{1}{2}$ at each time step, and it is integral, we get $\Delta_{l+2} = 0$. Therefore, $l + 2 = \max\{2\log K_i + i + 2\}$; this proves the lemma. □

THEOREM 2.9. *Canonical labelings for trees can be computed an* $O(\log n)$ *time, using* $O(n\log n)$ *processors.*

Proof. The algorithm consists of three majors steps, **as** summarized below:

1. Compute the label-time of each vertex.
2. Sort and order the children of each node up to their label-time value.
3. Compute the final ordering of the children by computing vertex labels using sorting.

Using Lemma 2.8, the label-time values for each node can be computed on or before its label-time. The label-time of a node is not passed to its parent until the actual time of the label-time value, thus preserving the invariant property that label-time values arrive at the actual time of the label-time value. Therefore, step 1 takes $O(\log n)$ time using $O(\log n)$ processors per node (nlogn in total).

In step 2, the children can be sorted at a node by their label-time values in $O(\log n)$ time using n processors. Finally, in step 3, the labels can be computed by sorting label values. **As** in Theorem 2.3, the timing analysis of Theorem 6.1 from [29] can be applied to give an $O(\log n)$ time bound. Again, using the analysis from the proof of Theorem 2.3 to step 2 of procedure Randomized$_1$ **Tree** Isomorphism, this algorithm requires at most n processors to achieve the $O(\log n)$ time bound. □

This motivates another generalization of *Parallel Tree Contraction* which will be used to compute the 3-connected components of a graph in $O(\log n)$ time, instead of $O(\log^2 n)$ time.

Consider *Asynchronous Parallel Tree Contraction,* **as** defined in Part 1 [29], applied to an ordered tree of unbounded degree, where the RAKE operation is restricted to removing **a constant proportion of consecutive leaves. In particular, assume** that **RAKE** replaces a run of length k by a run of length $\lfloor k/2 \rfloor$ in unit time. Thus, COMPRESS acts on chains, and **RAKE** acts on runs. Recall from Part **I** that a *chain* in a rooted tree is a sequence of vertices $v_1, \cdots, v_t$ such that v_{i+1} is the only child **of** v_i,

for $1 \leq i < t$. If the tree is undirected, a *chain* will be a sequence of vertices $v_1, \cdots, v_t$ such that v_{i-1} and v_{i+1} are the only neighbors of v_i, for $1 < i < t$. It is crucial that a vertex be processed under COMPRESS when it has one child that is not a leaf, or possibly two children that are leaves, a leftmost and, possibly, a rightmost child; i.e., one or two runs of length 1. This procedure is called *Parallel Tree Contraction with RAKE restricted to runs.*

THEOREM **2.10.** *Parallel Tree Contraction with RAKE restricted to runs requires only $O(\log n)$ applications to reduce a tree to a single vertex.*

3. Computing the 3-connected components. The main goal of this section is to give a new parallel algorithm for decomposing a graph into a tree of 3-connected components. To this end, we first discuss the decomposition of a graph into a tree of 2-connected components. We then discuss prior work on the decomposition of general graphs into their tree of 3-connected components, including a definition of brides and Hopcroft and Tarjan's use of virtual edges. Finally, we give our definition of the 3-connected components of a graph, and relate how to use *Parallel Tree Contraction* to find these components.

Two vertices v and w in an undirected graph $G = (V, E)$ are *k-connected* if there exist k paths in G from v to w which are pairwise vertex disjoint, except at their endpoints v and w. Thus, two vertices sharing k-edges are k-connected. The graph G is k-connected if every pair of vertices is k-connected.

Before giving our algorithm, which decomposes a connected graph into its tree of 3-connected components, we will discuss the decomposition of a connected graph into its tree of 2-connected components. This decomposition consists of three types of components. First, there are the proper 2-connected components. These are the subgraphs induced by a maximal subset of vertices which are pairwise 2-connected. Second, there are the articulation vertices or separating vertices. Finally, there are separating edges. The vertices of the tree consisting of 2-connected components are the components described above. An articulation vertex is adjacent to another component if it is contained in the component. Recently, Tarjan and Vishkin [36] have shown how to construct the 2-connected components of a graph in $O(\log n)$ time using a linear number of processors on a PRAM. These components form a tree where a component and a separating vertex are adjacent if the vertex is contained in the component. However, the 3-connected components are more difficult to define and seem to require a more sophisticated algorithm.

Hopcroft and Tarjan [16] give a precise algorithmic definition, which will be reviewed below, of the 3-connected components and show how any graph can be decomposed uniquely into a tree of 3-connected components. In the same paper, they also give a linear time algorithm for finding the tree of 3-connected components. Unfortunately, it is a highly sequential algorithm. A related distinct question is finding the maximal subsets of vertices of size greater than or equal to 2 which are pairwise 3-connected. These subsets shall be called the *3-sets* of G. Ja'Ja' and Simon [18] give an algorithm using $O(\log n)$ time and $n^{O(1)}$ processors for finding these 3-sets. There is a unique 3-connected graph associated with each 3-set. The proof and construction can be obtained by Lemma 3.1.

First, we will define the notion of a bridge. Let $C \subset V$. 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 it has 3 or more bridges or 2 or more nontrivial bridges. A *3-connected separating*

pair is a pair of vertices which is both 3-connected and a separating pair.

LEMMA 3.1. *If $C \subset V$ 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, 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 bconnected to all the elements of C. This contradicts the assumption that C is a (maximal) 3-set. □

Throughout the discussion of the 3-connected components, we let G be the underlying graph, which is assumed 2-connected. The tree of bconnected components consists of a tree of graphs called *components.* Two components are adjacent if they share an edge. These shared edges will not be edges from G, the original graph, but rather, from new edges called *virtual* edges. There will be exactly two copies of each virtual edge. Any vertex may appear in many components.

First, the graphs that will be the nodes in T will be described. The reader should be cautioned that, counter to intuition, the components are not always 3-connected graphs and separating pairs. The nodes of T are of three types: proper components, cycles, and m-bonds. The m-bonds lie between the components (proper components and cycles). They are precisely described below in Fig. 2, where the decomposition of a graph into components is shown. Note that the virtual edges are indicated by dotted lines.

- A *proper component* C is a simple 3-connected graph. C can be defined in terms of G as follows: the vertices of C consist of a 3-set S of size greater than or equal to 4 *(proper* 3-set). Two vertices of C share an edge in C if they shared one or more bridges in G. Note that C is simple; it has no multiple edges. An edge from x to y of C will be an original edge from G if x and y share exactly one trivial bridge; otherwise, the edge will be a virtual edge.
- A *cycle component* C is a simple cycle. C can be defined in terms of G as follows: the vertices of C are a maximal subset of the vertices S, such that the bridges of S in G form a simple cycle of size 3 or more, with possible pairs in S containing multiple bridges. As in the case of proper components, a unique trivial bridge e of S becomes an edge of C; otherwise, a virtual edge is formed.
- An *m-bond component* C is a graph on two vertices sharing two or more edges. C can be defined in terms of G as follows: x and y are the vertices of C if they are 3-connected and separating. There is one edge in C for each bridge of $\{x, y\}$ in G. If the bridge is trivial, the original edge in C can be used. Otherwise, a virtual edge is used. Note that 2-bonds have been introduced between two proper components or a proper component and a cycle component, which do not appear in the Hopcroft–Tarjan [16] definition.

We say a component is *associated* with another component if the two have a nonempty intersection.

We will now describe a parallel method for constructing the tree of components from the above three types of components. Our idea is to apply *Parallel Tree Contraction;* chains are not compressed, but, rather, every other component is removed from a chain. Since every other component on a path in the tree T is an m-bond, we can remove every other component on a chain by eliminating the *proper* and the *cycle* components. Thus, *proper* and *cycle* components associated with either zero, one, or

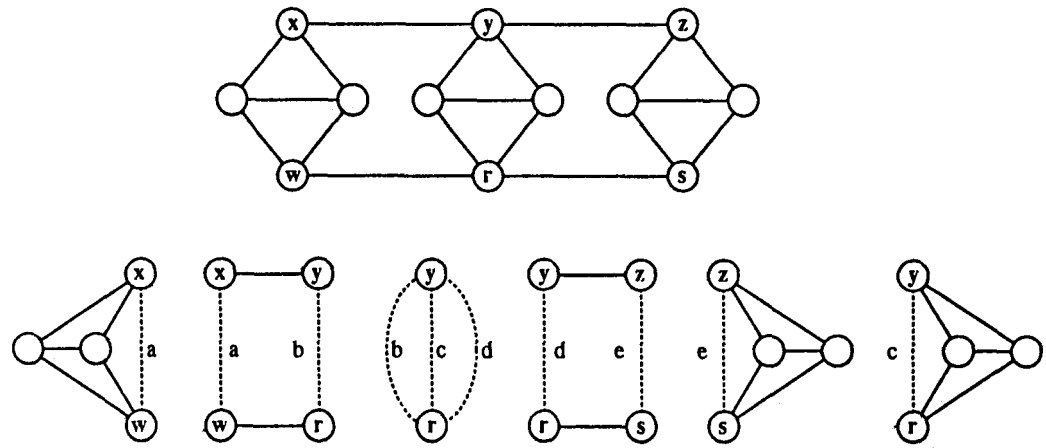

FIG. 2. *The decomposition of a gmph into its 3-connected components.*

two other components are removed, as are m-bond components associated with either zero or one other component. All these components are removed in unit time except for the cycle components, which may take as much as $O(\log n)$ time; we will show how to amortize the cost in such a way that the total time decomposition is still only $O(\log n)$.

Using the work of Ja'Ja' and Simon [18] we compute the 3-sets and their bridges, along with the separating pairs and their bridges, in $O(\log n)$ time using $n^{O(1)}$ processors. Note that they also determine which separating pairs are 3-connected.

Assume that G is stored in memory as an incidence matrix and that the following information is maintained: a list of proper 3-sets; a list of %connected separating pairs; a forest indicating which 3-connected separating pairs are contained in which proper 3-sets; and, for each 3-connected separating pair {z,y}, a list of edges associated with it, partitioned according to which bridge of $\{x, y\}$ they belong. An edge e from x to y is free if x and y are not 3-connected. Note that the free edges will belong to the cycle components. A list of free edges is also maintained.

Let T be the tree of components of G. As components are removed from G, G will no longer be connected. Therefore, intuitively, G should be a collection of 2-connected graphs. But for technical reasons, the connected components of G may not be 2-connected. This complication will be discussed when the **COMPRESS** part of the algorithm is discussed.

The discussion will begin with RAKE. Here, one must determine when a component becomes a leaf in T, at which time it is removed. Note that a component is a leaf if and only if it contains zero or one nontrivial bridge. The case when a component has exactly one nontrivial bridge will be discussed first. Note that a leaf component is a bridge to its parent. Thus, removing a leaf component decreases the number of nontrivial bridges by one. Suppose the component C is an m-bond with vertices $\{x, y\}$. Using a concurrent write and the fact that we maintain for C a list of all its bridges (and whether or not they are trivial), we are able, in unit time, to determine

that C is a leaf. To remove C from G, simply remove the trivial bridges of C from G, leaving x and y in G and adding to G a new virtual edge from x to y. The data structures are also updated as described above.

Suppose that C is a proper component. It is a leaf when it is associated with at most one 3-connected separating pair. Thus, one can test, in unit time, whether or not C is a leaf. If C is common to no 3-connected separating pairs, then simply ignore C, and do nothing to G or C. However, C is removed from all the other data structures. Suppose that C is common to one pair $\{x, y\}$. To remove C from G: (1) remove all vertices in C except x and y, (2) remove all edges with both end points in C except those between x and y, and (3) add a virtual edge in G from x to y.

To finish our discussion of RAKE, cycles will be considered. Suppose that C is a cycle. Since the vertices on C are unknown, they will be computed "on the fly." Suppose further that $(x_1, \cdots, x_k)$ are the vertices of a cycle component C in the order in which they appear on the cycle. The component C is a leaf if (1) each pair (x_i, x_{i+1}) for $1 \leq i < k$ contains exactly one bridge and that bridge is trivial; and (2) the pair (x_k, x_1) contains at least a trivial bridge. In other words, there exists an adjacent pair of vertices $\{x, y\}$ with a nontrivial bridge that consists of a path. The time required to remove each cycle component that is a leaf seems to require time logarithmic in the length of the its path to detect. We will show how to amortize this cost to achieve an overall time of $O(\log n)$. The edges (x_i, x_{i+1}) for $1 \leq i < k$ form a chain of free edges. Our idea is simply to "compress" these chains of free edges either by the deterministic or by the randomized methods discussed in [29]. In general, any chain can be compressed. Note that a chain of length two may be replaced by a chain of length one, which was formally not free, but it shall be considered free anyway. In this case, the cycle C has been "compressed" to a cycle of size two, a free edge common to a 3-connected separating pair {z,y}, and a virtual edge from x to y.

Thus, RAKE for cycles consists of compressing chains and removing free edges associated with a 3-connected virtual edge, and then replacing them with a new virtual edge. Other than this timing analysis, we have described RAKE.

The COMPRESS operation is very similar to RAKE. Here, each *proper* and **cycle** component associated with exactly two m-bonds is removed. Suppose that C is a *proper* component associated with 3-connected separating pairs $\{x, y\}$ and $\{z, w\}$. If x, y, z, and w are distinct, then the construction is very similar to the RAKE case. If, on the other hand, y = w, the situation is slightly more complicated, since simply removing the edges of C will not separate G. To remove C from G: (1) remove all vertices in C except x, y, z, and w; (2) remove all edges with both end points in C except those between x and y or between z and w; and (3) add a virtual edge in G from x to y and one from z to w.

Suppose C is as described above, except that it is a cycle component. C is removed only when it is a four-cycle component for the case when x, y, z, and w are distinct, or a three-cycle component for the case when y = w.

CONTRACT decomposes G into a tree T of 3-connected components after $O(\log n)$ applications. CONTRACT as defined (at least for the sake of analysis) can be viewed as simply CONTRACTION on trees of unbounded degree where RAKE is performed only by combining consecutive children. A case of CONTRACTION very similar to this was analyzed in Theorem 2.10 and shown to require only $O(\log n)$ steps.

Thus, G can be decomposed into a tree of 3-connected graphs, simple cycles, and m-bonds in $O(\log n)$ time using $n^{O(1)}$ processors. This can be stated in the following theorem.

THEOREM 3.2. *The tree* of *3-connected components* is *constructible an* $O(\log n)$ *tame, using* $n^{O(1)}$ *processors.*

Note that decomposition has been described only where the graph is 2-connected. In general, one must first decompose the graph into a tree of 2-connected components, which will consist of isolated vertices and 2-connected graphs. Second, one must further decompose a 2-connected graph into a tree of %connected components.

Ja'Ja' and Simon [18] tested whether or not a 3-connected graph is planar and, if it is, it constructs its planar embedding. However, the construction of a planar embedding **for** general planar graphs was an open question.

The next section shows how to construct the embedding of a planar graph given the tree of 3-connected components, and how to construct the embedding of each component by viewing it as a tree contraction problem. In this section, we will also define what we mean by "oriented embedding," and will show how to construct planar embeddings that will be used in isomorphism testing.

4. Graph embeddings and some applications. We will use the following combinatorial definition of an embedding, which is amenable to implementation on a machine.

DEFINITION 4.1. Let $G = (V,E)$ be an undirected graph. Two ***darts***, (x, y) and (y, x), **are** associated with each edge, $e = \{x, y\}$. 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 permutes 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. See [26], for example. In Fig. **3** we give **a** small example.

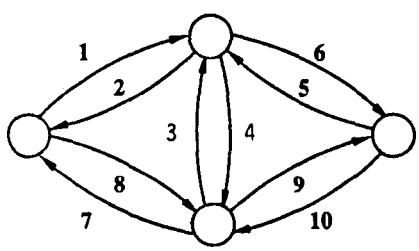

FIG. 3. *A graph with four vertices embedded in the plane. The permutation that determines the orientation at the vertices* $w\phi = (1\,8)(2\,6\,4)(3\,9\,7)(5\,10)$, *written in cycle notation. The reflection of the edges is* $R = (1\,2)(3\,4)(5\,6)(7\,8)(9\,10)$ *and face boundary written as a permutation is* $\phi^* = (1\,6\,10\,7)(2\,8\,3)(4\,9\,5)$.

This definition of a combinatorial embedding is similar to ones described in [10] and is sometimes called an Edmonds embedding; see also [26]. The importance **of** this definition of embedding is that it **is** both very simple to understand and easy to represent on a machine. **For** example, the faces are given by the permutation $\phi^* = \phi \cdot R$. The orbits of ϕ^* are the ***faces*** of the embedding. Also note that a permutation **is stored as an array;** thus, most operations on the permutation can be performed in constant time using a linear number **of** processors.

Ja'Ja' and Simon [18] give a parallel algorithm which constructs a planar embedding of a triconnected planar graph as defined above (note that they call this *planar mesh embedding.* They also construct a straight line embedding, which they call a barycentric embedding, which we do not use).

Using *Parallel Tree Contraction* proves the following theorem.

THEOREM 4.2. *Given the planar embeddings of the 3-connected components of a graph G, one can compute a planar embedding of G in $O(\log^2 n)$ time, using $O(|V|)$ processors.*

Proof. As described in Part 1 of this paper [29], *Parallel Tree Contraction* can be run "backwards" in an expansion mode which is called *Parallel Tree Expansion.* Here the 3-connectivity algorithm is run in the expansion mode. Thus, one initially starts with a collection of isolated components. The embedding of the isolated graphs is simply the embedding of the individual components. The inverse operation to both RAKE and COMPRESS, in this case, is simply combining two embedded graphs, T and T', by identifying two copies of a virtual edge $\{x, y\}$. The order in which the identification is performed is determined by *Parallel Tree Contraction.* Thus, the only procedure that needs to be shown is how to obtain the embedding for the new graph. Suppose embeddings of T and T' are both common to a virtual edge $e = (x, y)$. Here, the cyclic permutation of T at x is combined with the cyclic permutation of T' at x, which is done by determining a face F of the embedding of T which contains both x and y, and then determining a face F' of the embedding of T' which contains both x and y. The new cyclic order around x will begin by enumerating the darts of x in T as they appear in the embedding of T, starting with the dart in F, and then enumerating the darts of x in C as they appear in the embedding of T', starting with the dart in F'. At the same time, the cyclic permutations of T are combined at y, and the cyclic permutations of T' are combined at y in the same way (see Fig. 4).

To see that this construction can be performed in unit time we write out the permutation explicitly. Since T_1 and T_2 are disjoint graphs; we view them as having a single embedding ϕ. Let $\phi^* = \phi \cdot R$ be its dual. It will suffice to show how to construct the dual embedding $\widehat{\phi^*}$ for the identified graph. For simplicity we leave both copies of the virtual edge in the graph and embed them as parallel edges. The parallel edges can at a later time be removed. Let e_1 be an arc in T_1 from x to y and e_2 be an arc in T_2 from y to x. The dual embedding is as defined below:

$$\widehat{\phi^*} = \begin{cases} e_2 & \text{if } e = e_1, \\ e_1 & \text{if } e = e_2, \\ \phi^*(e_2) & \text{if } \phi^*(e) = e_1, \\ \phi^*(e_1) & \text{if } \phi^*(e) = e_2, \\ \phi^*(e) & \text{otherwise.} \end{cases}$$

□

This gives the following corollary.

COROLLARY 4.3. *A planar embedding of a planar graph with n vertices is constructible in $O(\log^2 n)$ time, using $n^{O(1)}$ processors.*

4.1. Canonical forms of oriented graphs. Whitney [39] has shown that every 3-connected planar graph has exactly two planar embeddings: an embedding ϕ and its reflection ϕ^{-1}. Ja'Ja' and Simon [18] have shown that a planar embedding can be constructed in $O(\log^2 n)$ time on a PRAM for a 3-connected planar graph. 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

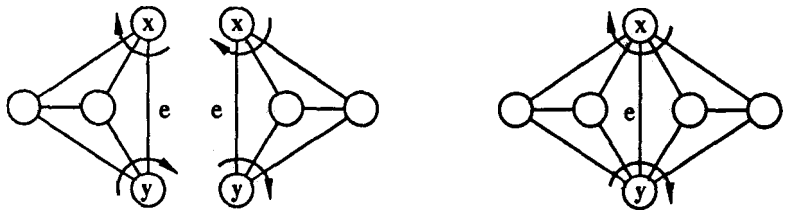

FIG. 4. *Combining the embeddings of two components to get a common embedding.*

both adjacency and orientation: $R'f = fR$ and $\phi' f = f\phi$. Using Whitney'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 embedded graph is determined by the image of a single dart. Given a sequence of k numbers, $u = (u_1, \cdots, u_k)$, and a dart e, there exists a unique path of length k, $e = e_0, \cdots, e_k$, where $e_i = \phi^{u_i} R(e_{i-1})$ for $1 \leq i \leq k$. Note that the length is the number of vertices on the path, and no is the number of edges. Given a path of darts, a unique sequence of integers can be constructed by choosing the minimum $u_i \geq 0$ such that $e_i = \phi^{u_i} R(e_{i-1})$. Next, we will show how to compute canonical sequences that will be used to compute the canonical forms for embedded graphs.

THEOREM 4.4. *Canonical numbering for oriented graphs is computable in $O(\log n)$ time, using $n^{O(1)}$ processors.*

A canonical form $M(e)$ for each dart e can be constructed in (G, ϕ). One simply picks the lexicographically least such form. For each dart, $e' \neq e$, the lexicographically least number sequence over the shortest paths from e to e' are found. Suppose that the graph G has d darts. Consider a d x d matrix where each entry is a number sequence or is blank. Here, the basic scalar operations will be *lexicographically minimum* and *concatenation,* which replace the operations, + and x. Initially, one starts with the matrix containing all paths of length two by storing a sequence of numbers of length one. A matrix product over *minimum* and *concatenation* can be computed in $O(1)$ time using $d^{O(1)} = n^{O(1)}$ processors by Lemma 2.8. Computing $O(\log n)$ iterated powers of this matrix, up to the d power of the original matrix, yields the lexicographically minimum of all shortest paths between all pairs of vertices. Thus, a canonical matrix $M(e)$ is obtained for each dart e in (G, ϕ). The minimum canonical matrix $M(e)$ (under lexicographical 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 adjacent matrices for the reflection (G, ϕ^{-1}) and computing the minimum over the larger set of matrices, canonical forms for embedded graphs have been constructed 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 $n^{O(1)}$ PRAM processors, the following theorem is derived from the above.

THEOREM 4.5. *Canonical numbering of 3-connected planar graphs can be done in $O(\log^2 n)$ time using $n^{O(1)}$ PRAM processors.*

5. Reducing the problem of finding canonical forms of planar graphs to the 3-connected case. In this section we give an $O(\log n)$ time reduction from finding canonical forms for general graphs to that of finding canonical forms for 3-connected graphs.

The term "computing canonical forms" means that an oracle accepts as input a 3-connected graph with labels on its darts and vertices and returns an incidence matrix unique up to isomorphism; i.e., it returns canonical linear ordering of the vertices. We also assume that there is a list of new labels that can be added to the darts or vertices.

By using the methods in the last section, one can find up to isomorphism a unique decomposition of a graph into a tree of 3-connected components. In this section, the 3-connected components are simply called "components." Two components are related if one identifies either (1) a virtual edge with orientation (a dart) in one with a virtual edge with orientation in the other, or (2) a vertex in one with a vertex in the other. We will discuss the case where the identifications are edges; i.e., the graph is 2-connected. The general case is a straightforward generalization.

Recall that not all components in a tree of 3-connected components are 3-connected; in particular, they can be either a 3-connected graph, a simple cycle, an m-bond, or an isolated vertex. Canonical forms for these latter graphs can easily be constructed in $O(\log n)$ time.

LEMMA 5.1. *The canonical form for labeled cycles and m-bonds can be constructed in $O(\log n)$ time using $n^{O(1)}$ processors.*

A node of maximum height (at the center of the tree) in a tree of 3-connected components can be found in $O(\log n)$ time by tree contraction, [29]. If the center of the tree is an edge, simply introduce a 2-bond, which will become the center of the tree, as a new component. Thus, without loss of generality, we may assume that the tree is rooted at either a 3-connected component, a virtual edge, or a 2-bond.

Since the rooted tree of 3-connected components is unique up to isomorphism, the vertices shall be ordered into blocks according to the component to which they belong. The separating pair is in the same block with the parent component. The blocks are ordered in postorder (see [36]). However, the children of a component must first be ordered. As in our construction for canonical orderings for regular trees, children will be first ordered at the time when labeled. The characteristic that distinguishes this from a regular tree case is the fact that the children are coupled to their parent by an edge and not a vertex. Thus, more information about the children must be passed to the parent.

Let C be a component and $e = (x, y)$ be the virtual edge of C common to its parent. The edge e is written as two darts d_1 and d_2 (the reverse of d_1). If C is a leaf and a proper component, then, by labeling either d_1 or d_2 with a new label, one gets two labels, L_1 and L_2, respectively, for C. Note that $L_1 = L_2$ if and only if there is an automorphism sending d_1 to d_2. Thus, RAKE is implemented in a Straightforward way: (1) compute the labels L_1 and L_2, (2) use the label of each leaf C to label the corresponding darts in the parent of C, and (3) remove C. These labels for C also give us the ordering of the vertices in C, excluding $\{x, y\}$. If $L_1 > L_2$, then use the ordering from L_1; the case is similar if $L_2 > L_1$. On the other hand,

if they are equal, then both orderings are the same, and it does not matter which one is picked. This completes the discussion of RAKE. Note that this computation of RAKE can be executed in unit time, given an oracle for generating the labels L_1 and L_2. COMPRESS will be discussed next.

Let C be a component of degree two where darts e_1 and e_2 are common to the parent and darts d_1 and d_2 are common to the only child. Using two new labels, L and L', assigning L to either e_1 or $e2$, and assigning L' to either d_1 or d_2 yields four labelings of C. Use the labeling with maximum value to determine the order of the vertices in C, excluding the end vertices of e. As before, if two labels are equal, then C has a symmetry; either order is the same up to isomorphism. This completes our discussion of COMPRESS.

It is important to point out that we have not determined where, in the final ordering, a given vertex was mapped, since finding this map was not required. This lack of information occurred when one of several orderings for a given component was arbitrarily picked in COMPRESS, and when the children of a component were simply sorted by label. One can determine up to a permutation of order two the ordering of components by using a tree expansion phase [29].

To compute the image of each vertex in the new ordering, it will suffice to determine the orientation induced on the virtual edges by the new ordering; i.e., is a given virtual edge left alone or is it reflected in the new ordering? COMPRESS will be discussed here (the case of RAKE is very similar). Let C be a component with two virtual edges e and d. The possible symmetries consist of reflecting e and independently reflecting d, the Klein 4 group K_4. The actual symmetries will be one of five possible subgroups. Thus, the canonical orderings will be a coset of one of these groups. There are thirteen such cosets, which can be determined by using a parallel call to the oracle for proper components (by applying Theorem 4.4) or which can be determined directly for cycles or m-bonds (by applying Lemma 5.1).

To implement COMPRESS, one need only compute the coset of canonical orderings for a consecutive pair of components from the coset of canonical orderings for each component. Let C and C' be two consecutive components of degree two with virtual edges, d, e, and f, respectively. Further, let A and B be the cosets of canonical ordcrings of C and C', respectively. Note that A acts on $\{d,e\}$ and B acts on $\{e,f\}$; one wants to return an appropriate coset acting on $\{d,f\}$. If the natural intersection is not empty, it will be returned as the coset of the canonical ordering for C and C'. It will be empty when A and B fix e in opposite orientations. In this case, the cosct of the canonical orderings for C and C' consists of a cross-product pair. One acts on d according to A and acts on f according to B. Thus, a method for computing the coset of canonical orderings for the virtual edges of $C \cup C'$ has been presented which uses $O(\log n)$ time and $n^{O(1)}$ processors.

In summary, the CONTRACTION phase consists of the followiiig steps:

1. Compute the canonical labels for all components with degree one or two and determine the coset of canonical orderings on their virtiial edges.
2. For leaves, pass the canonical label to the parent.
3. For chains, combine pairs of components as described above, coniputing both canonical labels and cosets of canonical orderings.

Note that there will be missing cosets when we execute chain contraction. After the tree of components has been reduced to a single component, we perform a tree expansion phase as described in [29] to compute the missing cosets from this further information obtained. Each step can be executed in unit time and thus, by the analysis

in [29], the total time is $O(\log n)$.

We have just given an $O(\log n)$ time reduction from finding canonical forms for general graphs to that of canonical forms for 3-connected components.

$O(\log^2 n)$ time, $n^{O(1)}$ processor algorithms used for finding canonical forms for 3-connected graphs have already been presented. This reduction implies an $O(\log^3 n)$ time, $n^{O(1)}$ processor algorithm that can be used to compute canonical forms for all planar graphs. We summarize our results as a theorem.

THEOREM 5.2. *The problem "Computing canonical forms for a general graph" is $O(\log n)$ time wing $n^{O(1)}$ processors reducible to the problem "computing canonical forms for its 3-connected components."*

6. Conclusion. Since the original writing of this paper, many other applications of *Parallel Tree Contraction* have been found. Similarly, many extensions, improvements, and simplifications of the work in this paper have been found. The basic parallel tree contraction can now be done on an EREW PRAM in $O(\log n)$ deterministic time using $n/\log n$ processors, [9], [21], [13], [1]. All of these algorithms use the fact that list-ranking can be performed optimally in deterministic time $O(\log n)$ on an EREW PRAM, [3], [9]. Very simple randomized algorithms for the list-ranking problem are also known, [5]. *Parallel Tree Contraction* can be performed optimally by a randomized algorithm on a parallel model that is more restrictive than an EREW PRAM [4].

In this paper, we restricted our attention to maximal subtree isomorphism. The more general problem for determining if one tree is a subtree of another was first addressed by Matula [22], who gave a polynomial-time algorithm for the problem. A randomized NC algorithm for this problem was given in [14] using the parallel tree contraction technique.

PTC has also been used for efficient parallel evaluation of arithmetic circuits. Prior to the parallel tree contraction technique, the best algorithms for the circuit problem used divide-and-conquer, [37]. Using PTC, one can evaluate circuits on-line in the same time and size as [37] achieved off-line [28], [23].

PTC can be used to design efficient parallel algorithms for problems where the tree is known only implicitly. Examples of such problems occur in the context-free language parsing, constructing Huffman codes, and optimal binary search trees. See [33] for an example of a divide-and-conquer algorithm for such problems and see [6] for a PTC-based approach.

Other applications include: testing triconnectivity of a graph [27], [11]; testing graph planarity [19]; finding separator for planar graphs [25], [12]; and finding algorithms for reducible flow graphs [30].

This is not an exhaustive list, and we apologize for the works which we have neglected to reference.

REFERENCES

[1] K. ABAHAMSON, N. DADOUN, D. K. KIRKPATRICK, AND T. PRZYTYCKA, *A simple parallel tree contraction algorithm (preliminary version)*, in Proc. 25th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, September/October 1987, pp. 624–633.

[2] M. AJTAI, J. KOMLOS, AND E. SZEMEREDI, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual Symposium on the Theory of Computing, 1983, pp. 1–9.

[3] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing (AWOC 88), June/July 1988,

Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, pp. 81–90.

[4] R. J. ANDERSON AND G. L. MILLER, *Optical communication for pointer based algorithms,* Tech. Report CRI 88-14, Department of Computer Science, University of Southern California, Los Angeles, CA, 1988.

[5] ——, *A simple randomized parallel algorithm for list-mnking,* Inform. Process. Lett., 33 (1990), pp. 269–273.

[6] M. ATALLAH, R. KOSARAJU, L. LARMORE, G. L. MILLER, AND S.-H. TENG, *Constructing trees in parallel,* in Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, June 1989, pp. 421–431.

[7] I. BAR-ON AND U. VISHKIN, *Optimal parallel generation of a computation tree form,* ACM Trans. Programming Languages and Systems, 7 (1985), pp. 348–357.

[8] R. COLE, *Parallel merge sort,* in Proc. 27th IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, October 1987, pp. 511–516.

[9] R. COLE AND U. VISHKIN, *Optimal parallel algorithms for expression tree evaluation and list ranking,* in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing (AWOC 88), June/July 1988, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, 1988, pp. 91–100.

[10] J. EDMONDS, *A wmbinatorial representation for polyhedral surfaces,* Amer. Math. Soc., 7 (1960), p. 646.

[11] D. FUSSELL, V. RAMACHANDRAN, AND R. THURIMELLA, *Finding triconnected components by local replacement,* in Proc. Internat. Conference on Automata, Languages and Programming, 1989, Springer-Verlag, pp. 379–393.

[12] H. GAZIT AND G. L. MILLER, *A parallel algorithm for finding a sepamtor in planar graphs,* in 28th IEEE Annual Symposium on Foundations of Computer Science, Los Angeles, CA, October 1987, pp. 238–248.

[13] H. GAZIT, G. L. MILLER, AND S.-H. TENG, *Optimal tree contraction in an EREW model,* in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., Plenum Press, New York, 1988, pp. 139–156.

[14] P. B. GIBBONS, R. M. KARP, G. L. MILLER, AND D. SOROKER, *Subtree isomorphism is in random* NC, in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing, (AWOC 88), June/July 1988, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, 1988, pp. 43–52.

[15] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers,* Fourth Edition, Oxford University Press, Oxford, U.K., 1959.

[16] J. E. HOPCROFT AND R. E. TARJAN, *Dividing a graph into triwnnected components,* SIAM J. Comput., 2 (1973), pp. 135–158.

[17] O. H. IBARRA AND S. MORAN, *Probabilistic algorithms for deciding equivalence of stmight-line programs,* J. Assoc. Comput. Mach., 30 (1983), pp. 217–228.

[18] J. JA'JA' AND J. SIMON, *Parallel algorithms in graph theory: Planarity testing,* SIAM J. Comput., 11 (1982), pp. 314–328.

[19] P. KLEIN AND J. REIF, *An efficient parallel algorithm for planarity,* J. Comput. System Sci., (1988), pp. 190–246.

[20] D. E. KNUTH AND L. TRABB PARDO, *Anaiysis of a simple factorization algorithm,* Theoret. Comput. Sci., 3 (1976), pp. 321–348.

[21] S. R. KOSARAJU AND A. L. DELCHER, *Optimal parallel evaluation of tree-structured computation by ranking (extended abstract),* in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing (AWOC 88), June/July 1988, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, pp. 101-110.

[22] D. W. MATULA, *Subtree isomorphism in $O(n^{5/2})$,* Ann. Discrete Math., 2 (1978), pp. 91–106.

[23] E. W. MAYR, *The dynamic tree expression problem,* in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., 1988, Plenum Press, New York, pp. 157–180.

[24] G. L. MILLER, *Riemann's hypothesis and tests for primality,* J. Comput. System Sci., 13 (1976), pp. 300-317.

[25] ——, *Finding small simple cycle separators for 2-connected planar graphs,* J. Comput. System Sci., 32 (1986), pp. 265–279.

[26] ——, *An additivity theorem for the genus of a graph,* J. Combin. Theory, Ser. B, 43 (1987), pp. 25–47.

[27] G. L. MILLER AND V. RAMACHANDRAN, *A new gmph triconnectivity algorithm and its parallelization (extended abstract),* in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, May 1987.

[28] G. L. MILLER, V. RAMACHANDRAN, AND E. KALTOFEN, *Efficient parallel evaluation of straight-line code and arithmetic circuits*, SIAM J. Comput., 17 (1988), pp. 687–695.

[29] G. L. MILLER AND J. H. REIF, *Parallel tree contraction Part 1: Fundamentals*, in Randomness and Computation, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 47–72.

[30] V. RAMACHANDRAN, *Fast parallel algorithms for reducible flow graphs*, in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., Plenum Press, New York, 1988, pp. 117–138.

[31] J. H. REIF AND L. G. VALIANT, A *logarithmic time sort for linear size networks*, in Proceedings of the 15th Annual ACM Symposium on Theory of Computing, Boston, 1983, pp. 10–16.

[32] J. B. ROSSER AND L. SCHOENFIELD, *Approximate formulas for some functions of prime numbers*, Illiois J. Math., 6 (1962), pp. 64–94.

[33] W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[34] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[35] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85.

[36] R. E. TARJAN AND U. VISHKIN, *Finding biconnected components and computing tree functions in logarithmic parallel time*, in Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science, FL, 1984, pp. 12–22.

[37] L. G. VALIANT, S. SKYUM, S. BERKOWITZ, AND C. RACKOFF, *Fast parallel computation of polynomials using few processors*, SIAM J. Comput., 12 (1983), pp. 641–644.

[38] U. VISHKIN, *Randomized speed-ups in parallel computation*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Washington D.C., April 1984, Association for Computing Machinery, pp. 230–239.

[39] H. WHITNEY, *A set of topological invariants for graphs*, American J. Math., 55 (1937), pp. 321–335.

[40] J. C. WYLLIE, *The complexity of parallel computations*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1981.