SUBTREE ISOMORPHISM IS IN RANDOM NC

Phillip B. Gibbons
University of California, Berkeley
Richard M. Karp
University of California, Berkeley
Gary L. Miller
University of Southern California
Danny Soroker
University of California, Berkeley*

December 2, 1987

Abstract

Given two trees, a guest tree G and a host tree H, the subtree isomorphism problem is to determine whether there is a subgraph of H that is isomorphic to G. We present a randomized parallel algorithm for finding such an isomorphism, if it exists. The algorithm runs in time $O(\log^3 n)$ on a CREW PRAM, where n is the number of nodes in H. Randomization is used (solely) to solve each of a series of bipartite matching problems during the course of the algorithm. We demonstrate the close connection between the two problems by presenting a log space reduction from perfect bipartite matching to subtree isomorphism. Finally, we present some techniques to reduce the number of processors used by the algorithm.

*The first, second, and fourth authors' address is Computer Science Division, University of California, Berkeley, CA 94720. The third author's address is Department of Computer Science, University of Southern California, Los Angeles, CA 90089. The first author was supported by the International Computer Science Institute (ICSI), Berkeley, California and by an IBM Doctoral Fellowship. The second author was supported in part by ICSI and NSF Grant #DCR-8411954. The third author was supported in part by NSF Grant #DCR-8514961 and by the Mathematical Sciences Research Institute (MSRI), Berkeley, California. The fourth author was supported by ICSI and by Defense Advanced Research Projects Agency (DoD) Arpa Order #4871, monitored by Space and Naval Warfare Systems Command under Contract #N00039-84-C-0089.

1 Introduction

A subtree of a tree T is any subgraph of T that is a tree. Given two (unrooted) trees, a guest tree G and a host tree H, the subtree isomorphism problem is to determine whether there is a subtree of H that is isomorphic to G. There is an $O(n^{2.5})$ sequential algorithm for this problem due to Matula[Mat78], where n is the number of nodes in H. In this paper, we present an $O(\log^3 n)$ time randomized algorithm for a CREW PRAM that exhibits the mapping between the trees, if such a mapping exists. We assume the word size of the PRAM is $c \log n$ for some constant c. With a few techniques to reduce the processor count, the algorithm uses $< \sqrt{n} \cdot P(n)$ processors, where $P(n)$ is the number of processors needed for *one* bipartite matching problem on n nodes using the fastest algorithm for bipartite matching to date[MVV87], for a total of $o(n^{4.9})$ processors. More precisely, let $M(n)$ be the best sequential arithmetic time bound for multiplying two $n \times n$ matrices. Then our algorithm uses $n^{2.5} M(n)/\log^3 n$ processors.

Our algorithm is based on Matula's sequential algorithm. The main obstacle to developing a fast parallel algorithm from Matula's algorithm is that its running time is proportional to the height of the tree. But by adapting the dynamic tree contraction algorithm of Miller and Reif[MR85], we show that subtree isomorphism is in random-NC (RNC). Dynamic tree contrac-

tion is one of two classic methods for achieving NC and RNC algorithms for problems involving potentially unbalanced trees; the other is recursively finding a vertex "1/3 – 2/3" separator for the tree[LT75]. In a complementary effort, Lingas and Karpinski[LK87] independently developed an RNC^3 algorithm for subtree isomorphism based on the latter method. In both algorithms, the randomization is used to perform the bipartite matching problems: if a (deterministic) NC algorithm is found for bipartite matching, then subtree isomorphism will also be in NC.

As in Matula's algorithm, we recast subtree isomorphism as a problem on limbs of G and H. A limb of a tree T is a subgraph of T rooted at a vertex u consisting of an edge $\{u, v\}$ of T, together with the connected component of $T - \{\{u, v\}\}$ which contains v. Let $T\langle u, v\rangle$ denote the limb of T defined by the root vertex u and the edge $\{u, v\}$. A rooted tree will be represented with each node (except the root) having an edge directed to its parent. Any edge $\{v, w\} \in T, w \neq u$, will determine a limb $T\langle v, w\rangle$, a subgraph of $T\langle u, v\rangle$, which we call a child limb of $T\langle u, v\rangle$ (see Fig. 1). If $\{t_1, t_2\}$ is the only edge incident to t_1 in T, then the limb $T\langle t_1, t_2\rangle$ contains all of T, and is denoted a root limb, and $T\langle t_2, t_1\rangle$ is denoted a leaf limb. A tree S is isomorphic to a tree T if and only if some root limb $S\langle s_1, s_2\rangle$ is isomorphic to some root limb $T\langle t_1, t_2\rangle$, where s_1 is mapped to t_1 and s_2 is mapped to t_2. This is called a limb imbedding of $S\langle s_1, s_2\rangle$ in $T\langle t_1, t_2\rangle$. The height of a limb $T\langle t_i, t_j\rangle$ denotes the maximum distance of any vertex of $T\langle t_i, t_j\rangle$ from the root vertex t_i.

Given two trees G and H, one can test whether there is subtree of H that is isomorphic to G as follows. First choose a root limb $G\langle g_1, g_2\rangle$ of G. G can be imbedded in H if and only if $G\langle g_1, g_2\rangle$ can be imbedded in some limb of H (typically not a root limb). To determine whether $G\langle g_i, g_j\rangle$ can be imbedded in $H\langle h_k, h_l\rangle$, one can apply the following theorem due to Matula:

Theorem 1 *[Mat78]: Let $G\langle g_i, g_j\rangle$ be a limb of a tree G and $H\langle h_l, h_m\rangle$ be a limb of a tree H. Consider a bipartite matching problem between the child limbs of $G\langle g_i, g_j\rangle$ and the child limbs of $H\langle h_l, h_m\rangle$, where there is an edge between child limbs $G\langle g_j, g_k\rangle$ and $H\langle h_m, h_n\rangle$ if and only if $G\langle g_j, g_k\rangle$ is limb imbeddable in $H\langle h_m, h_n\rangle$. Then there exists a matching that matches all the child limbs of $G\langle g_i, g_j\rangle$ if and only if $G\langle g_i, g_j\rangle$ is limb imbeddable in $H\langle h_l, h_m\rangle$.*

In this way, an instance of the subtree isomorphism problem can be solved using a series of bipartite matchings by starting at limbs of height 1 and progressing up to the root of the tree.

In order to achieve polylogarithmic running time, we will apply dynamic tree contraction to the guest tree. As G is contracted, each remaining nonroot node represents larger and larger subtrees of the original guest tree. For the *rake* operation, we set up and solve bipartite matching problems for each leaf limb in G, as in theorem 1. The difficulty lies in defining a suitable *compress* operation which must combine two adjacent G nodes (limbs) before it is known where in H their respective remaining child limb can be imbedded. For each limb with exactly one child limb, we compute and maintain a set of conditionals defining its imbeddability in H as a function of the as-yet-unknown imbeddability of its one child limb. As part of a compress operation, such conditionals for a limb can be readily composed with the conditionals for its child. Once G has been contracted, we can determine the isomorphic mapping between G and H by expanding G back to its original size, computing the map for each G node as it is added back to the tree.

To reduce the number of processors, we will apply two bipartite matching algorithms: one for the *decision* problem (determine if a perfect matching exists) while contracting the tree, and one for the *search* problem (find the edges in a perfect matching) while expanding the tree. Further processor savings are achieved by solving groups of related matching problems at once. Finally, we show that perfect bipartite matching and subtree isomorphism are mutually NC reducible by presenting a log space reduction from the former to the latter.

Miller and Reif[MR85] use dynamic tree contraction to develop NC algorithms for the related problems of tree isomorphism, canonical labels for trees, and canonical labels for all subtrees. The latter problem assigns labels to all nodes in a rooted tree such that two nodes u and v have the same label if and only if the *maximal* subtree rooted at u is isomorphic to the *maximal* subtree rooted at v. This differs from the subtree isomorphism problem, in which the subtrees of H are not necessarily maximal.

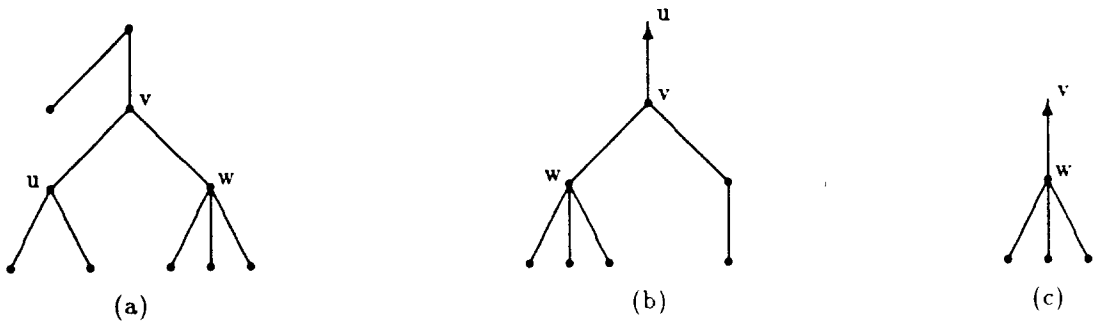


Figure 1: (a) A tree T. (b) Limb $T\langle u, v\rangle$. (c) Limb $T\langle v, w\rangle$, a child limb of $T\langle u, v\rangle$.

Since our algorithm makes use of the work of Matula[Mat78], Miller and Reif[MR85], and Mulmuley, Vazirani, Vazirani[MVV87], we first review their algorithms in the next section. (Readers familiar with these algorithms may skip to section 3). In section 3, we present our algorithm. Section 4 describes how to reduce the number of processors used, while section 5 presents a log space reduction of bipartite matching to subtree isomorphism.

2 Background work

We begin this section on background work with Matula's sequential algorithm for subtree isomorphism.

Algorithm A[Mat78]: Given a tree H on n_H vertices $\{h_1, \ldots, h_{n_H}\}$ and a tree G on n_G vertices $\{g_1, \ldots, g_{n_G}\}$, this algorithm determines if there is a subtree of H isomorphic to G.

A1. Select a root limb $G\langle g_1, g_2\rangle$ of G and root G accordingly. Order the $(n_G - 1)$ limbs of $G\langle g_1, g_2\rangle$ (one for each directed edge in $G\langle g_1, g_2\rangle$) by non-decreasing height.

A2. Let **Imbed**[,] be an $(n_G - 1) \times (2n_H - 2)$ matrix, with one entry for each pair $(G\langle g_i, g_j\rangle, H\langle h_k, h_l\rangle)$ of limbs associated with the rooted $G\langle g_1, g_2\rangle$ and the nonrooted H. During the course of the algorithm, **Imbed**$[\langle g_i, g_j\rangle, \langle h_k, h_l\rangle]$ = "imbeddable" if and only if $G\langle g_i, g_j\rangle$ is found to be limb imbeddable in $H\langle h_k, h_l\rangle$. Initially, mark all limbs in $G\langle g_1, g_2\rangle$ of height 1 as "imbeddable" in all H limbs. Mark all other entries as "don't know".

A3. Let h step by 1 from 2 through the height of the limb $G\langle g_1, g_2\rangle$. For each limb $G\langle g_i, g_j\rangle$ of height h and each limb $H\langle h_l, h_m\rangle$, set up and solve a bipartite matching problem as in theorem 1, based on the values in the **Imbed** matrix for the respective child limbs. Set **Imbed**$[\langle g_i, g_j\rangle, \langle h_l, h_m\rangle]$ to "imbeddable" or "unimbeddable" accordingly.

A4. If there is a limb $H\langle h_k, h_l\rangle$ such that **Imbed**$[\langle g_1, g_2\rangle, \langle h_k, h_l\rangle]$ = "imbeddable", then there is a subtree of H isomorphic to G. If not, then no such subtree exists.

Theorem 2 *[Mat78]: Given two trees G and H, algorithm A determines if there is a subtree of H isomorphic to G.*

Algorithm A can be modified to exhibit the mapping between G and a subtree of H (if any) as follows. Retain the solutions to the matching problems solved by the algorithm as it progresses up the tree. Then retrace the algorithm from the root of G back down to height 1, using these solutions to determine the actual limbs matched. With some tricks, Matula is able to reduce the running time of his algorithm to $O(n_H^{3/2} n_G)$.

Let T be a rooted tree with n nodes and root r. Miller and Reif[MR85] define two generic operations on T such that at most $O(\log n)$ applications of these operations are needed to contract

T to a single node. At step t, the first operation, called rake, removes all leaves (nodes with in-degree zero) from the tree which resulted after step $t-1$. Let a chain be a maximal sequence of nodes $v_1, \ldots, v_k$ in T where v_{i+1} is the only child of v_i for $1 \leq i < k$, v_1 is not the root, and v_k has exactly one child and that child is not a leaf. The second generic operation, called compress, compresses all chains in the current tree into chains a constant factor shorter. The compress operation determines the parity of each node v_i in its chain. If v_i is of even parity, then it is connected to its grandparent v_{i-2}, and its parent v_{i-1} is deleted from the chain. (This implementation of the compress operation is sufficient for our algorithm. See [MR85] for improved versions of the compress operation). These two operations together form one contract phase. The power of a contract phase is summed up in the following theorem.

Theorem 3 *[MR85]: A rooted tree with n nodes can be contracted to a single node in $O(\log n)$ contract phases.*

Consider a typical application of dynamic tree contraction: there is a problem on trees where the sequential algorithm to solve the problem starts at the leaves of the tree and works up to the root. Initially, the "values" of the leaves are known. Inductively, the value of a node can be computed once the values of all its children are known. In dynamic tree contraction, one defines a suitable "partial-value" for a node for which values are known for only some of its children. As the tree is being contracted, values are maintained for all leaves and partial-values are maintained for all other nodes.

In defining a contract phase, one distinguishes between 4 types of (nonroot) nodes, based on the number and type of their children.

1. The node has no children, i.e. it is a leaf. As part of the rake operation, this node is deleted from the tree.

2. The node has children and they are all leaves. As part of the rake operation, this node computes its new value from its partial-value and the values of its children.

3. The node has some children which are leaves and some which are not leaves. As part of the rake operation, this node computes its new partial-value from its partial-value and the values of its leaf children.

4. The node has children, but none of them are leaves. There are two subtypes to consider:

   (a) The node has > 1 (nonleaf) children. This node sits out this phase.

   (b) The node has exactly 1 (nonleaf) child. As part of the compress operation, this node may either (i) be deleted, or (ii) combine partial-values with its parent (which gets deleted) to get a new partial-value while making its grandparent be its new parent (one application of *pointer jumping*).

We now sketch the Mulmuley, Vazirani, Vazirani randomized algorithm[MVV87] for finding a perfect matching in a bipartite graph. They first assign random integer weights to the edges of the graph in order to make the minimum-weight perfect matching of the resulting graph unique (with probability $\geq 1/2$). Then they apply algebraic techniques to enable each processor to independently determine if its particular edge is in this unique matching. Let $\mathbf{det}(A)$ denote the determinant of a square matrix A, $\mathbf{adj}(A)$ denote the adjoint of A, and $\boldsymbol{A_{ij}}$ denote the submatrix (minor) of A obtained by removing the i^{th} row and j^{th} column. Their algorithm is as follows:

Algorithm B[MVV87]: Given a bipartite graph $A = (U, V, E)$, where $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, with at least one perfect matching, this algorithm finds a perfect matching in A with probability $\geq 1/2$.

B1. Assign random weights w_{ij} to the edges of A, chosen uniformly and independently from the range $[1, 2n]$.

B2. Consider the weighted adjacency matrix M for A, where the $(i,j)^{\text{th}}$ entry is $2^{w_{ij}}$ if $(u_i, v_j) \in E$, and 0 otherwise. Compute $\det(M)$, and obtain w, the highest power of 2 which divides $\det(M)$. This will be the weight of the perfect matching found in A.

B3. Compute $\text{adj}(M)$; its $(j,i)^{\text{th}}$ entry will be $\det(M_{ij})$.

B4. For each edge $\{u_i, v_j\}$ in parallel do:

B5. Compute $\frac{\det(M_{ij})2^{w_{ij}}}{2^w}$;

B6. If this quantity is odd, include $\{u_i, v_j\}$ in the matching.

end;

Theorem 4 *[MVV87]: Given a bipartite graph $A = (U, V, E)$ with at least one perfect matching, algorithm B finds a perfect matching in A with probability $\geq 1/2$.*

The resource requirements of algorithm B are bounded by the time and processors needed to compute $\det(M)$ and $\mathrm{adj}(M)$. They use Pan's algorithm[Pan85][GP85a] for inverting integer matrices, which computes $\det(M)$ and $\mathrm{adj}(M)$ in order to compute M^{-1}. Pan's algorithm takes time $O(\log^2 n)$ and $nlM(n)\log n \log\log n$ bit operations to invert (with high probability) an $n \times n$ matrix whose entries are l-bit integers. (There is a deterministic version which uses n times as many bit operations). Thus algorithm B takes $O(\log^2 n)$ time and $n^2 M(n)\log\log n/\log n$ processors, i.e. $o(n^{4.4})$ processors, since the entries of M are $(2n)$-bit integers.

3 The subtree isomorphism algorithm

As in Matula[Mat78], we first select a root limb $G\langle g_1, g_2\rangle$ of G and root G accordingly. This determines a set of $n_G - 1$ limbs associated with the rooted G. As the algorithm proceeds, it contracts G using suitably defined rake and compress operations. We will associate the limb $G\langle g_i, g_j\rangle$ with its second vertex g_j, and name the limb $G\langle \star, g_j\rangle$ to reflect the fact that the parent node of g_j may be changing throughout the algorithm as a result of compress operations. The host tree H is not rooted and thus has $2(n_H - 1)$ limbs to consider. As the algorithm does not alter H, we will name its limbs with two vertices, e.g. $H\langle h_k, h_l\rangle$.

As G is contracted, the limbs remaining will represent larger and larger subtrees of the original tree. Let $G\langle g_p, g_q\rangle$ be the limb in the original tree that $G\langle \star, g_j\rangle$ currently represents. We will call such a $G\langle g_p, g_q\rangle$ the original-limb of $G\langle \star, g_j\rangle$. Initially, and until the first compress operation involving g_j, $G\langle g_i, g_j\rangle$ will be the original-limb of $G\langle \star, g_j\rangle$. As a result of a compress operation, the new original-limb of $G\langle \star, g_j\rangle$ will be the current original-limb of its parent (the parent gets deleted). As G is contracted, we will say that a limb $G\langle \star, g_j\rangle$ is imbeddable in a particular H limb if and only if its original-limb is imbeddable in the H limb. The algorithm continues to contract G until it consists of just 1 limb (2 nodes), which will by then represent all of the original guest tree.

After initialization and throughout the course of the algorithm, we will maintain the following invariants on the G limbs. After t contract phases, the following properties hold for limbs remaining in the tree.

Invariants I:

- The value of any leaf limb of G has been computed, i.e. the set of H limbs in which (the original-limb of) the leaf limb can be imbedded has been determined.
- The partial-value of any limb $G\langle \star, g_i\rangle$ with exactly one child limb has been computed. The partial-value gives the set of H limbs in which (the original-limb of) $G\langle \star, g_i\rangle$ can be imbedded under certain conditions on (the original-limb of) its child limb. Let $G\langle \star, g_j\rangle$ be the child limb of $G\langle \star, g_i\rangle$ in the current G tree. A limb $H\langle h_m, h_n\rangle$ is in the conditional-set of the pair $G\langle \star, g_i\rangle$ and $H\langle h_k, h_l\rangle$ if and only if $\{G\langle \star, g_j\rangle$ is imbeddable in $H\langle h_m, h_n\rangle\}$ implies $\{G\langle \star, g_i\rangle$ is imbeddable in $H\langle h_k, h_l\rangle\}$. A short hand notation for this situation is "$H\langle h_k, h_l\rangle$ if $H\langle h_m, h_n\rangle$". Note: As a result of compress operations, $H\langle h_m, h_n\rangle$ may not be a child limb of $H\langle h_k, h_l\rangle$. The partial-value of a limb with exactly one child is the collection of its $2(n_H - 1)$ conditional-sets.
- The trivial partial-value of any limb $G\langle \star, g_i\rangle$ with > 1 child limbs has been computed, i.e. a list of children raked in the first t steps from $G\langle \star, g_i\rangle$ has been retained, along with the values of each such child.

In defining the $(t+1)^{\text{th}}$ contract phase, we distinguish between 4 types of (nonroot) nodes, as in section 2. In contrast to the outline of a contract phase presented in section 2, our contract

waits until a limb has only one child before it computes a nontrivial partial-value for that limb. Recall that we equate a limb $G\langle\star, g_i\rangle$ with its associated node g_i.

1. The limb $G\langle\star, g_i\rangle$ has no child limbs, i.e. it is a leaf. As part of the rake operation, g_i is deleted from G, and its value is saved with its parent.

2. The limb $G\langle\star, g_i\rangle$ has child limbs and they are all leaves. We distinguish between two subtypes:

   (a) The limb has > 1 (leaf) child limbs. As part of the rake operation, the node g_i computes its value by considering the values of all children raked from this node, this phase and previous phases. As in theorem 1, determine the set of H limbs in which $G\langle\star, g_i\rangle$ can be imbedded using bipartite matching.

   (b) The limb has exactly 1 (leaf) child limb $G\langle\star, g_j\rangle$. As part of the rake operation, g_i computes its value by plugging the child's value into its partial-value (conditional-set). For example, given the conditional "$H\langle h_k, h_l\rangle$ if $H\langle h_m, h_n\rangle$" and the fact that $G\langle\star, g_j\rangle$ is limb imbeddable in $H\langle h_m, h_n\rangle$, conclude that $G\langle\star, g_i\rangle$ is limb imbeddable in $H\langle h_k, h_l\rangle$.

3. The limb $G\langle\star, g_i\rangle$ has some child limbs which are leaves and some which are not leaves.

   (a) The limb has > 1 nonleaf child limbs. The node g_i sits out this phase.

   (b) The limb has exactly 1 nonleaf child limb $G\langle\star, g_j\rangle$. As part of the rake operation, g_i computes its conditional-set by considering the values of all children raked from this node, this phase and previous phases. For each limb $H\langle h_k, h_l\rangle$, set up a series of bipartite matching problems, one for each of its child limbs. For child limb $H\langle h_m, h_n\rangle$, solve a bipartite matching problem between the child limbs of $G\langle\star, g_i\rangle$ and the child limbs of $H\langle h_k, h_l\rangle$, as in (2a) above, except exclude child limbs $G\langle\star, g_j\rangle$ and $H\langle h_m, h_n\rangle$ from the matching problem. If there exists a matching that matches all such child limbs of $G\langle\star, g_i\rangle$, then $G\langle\star, g_i\rangle$ is limb imbeddable in $H\langle h_k, h_l\rangle$ on the condition that $G\langle\star, g_j\rangle$ is limb imbeddable in $H\langle h_m, h_n\rangle$, and include "$H\langle h_k, h_l\rangle$ if $H\langle h_m, h_n\rangle$" in the conditional-set for g_i.

4. The limb has child limbs, but none of them are leaves.

   (a) The limb has > 1 (nonleaf) child limbs. The node g_i sits out this phase.

   (b) The limb has exactly 1 (nonleaf) child limb. As part of the compress operation, g_i may either (i) be deleted, or (ii) combine partial-values with its parent (which gets deleted) while pointer jumping to point to its grandparent. Any node on a chain must have only one child, so partial-values can be combined by composing the conditionals associated with g_i and its parent. For example, "$H\langle h_j, h_k\rangle$ if $H\langle h_l, h_m\rangle$" in the conditional-set for the parent and "$H\langle h_l, h_m\rangle$ if $H\langle h_n, h_o\rangle$" in the conditional-set for $G\langle\star, g_i\rangle$, results in "$H\langle h_j, h_k\rangle$ if $H\langle h_n, h_o\rangle$" in the conditional-set for $G\langle\star, g_i\rangle$. This latter conditional-set reflects the fact that $G\langle\star, g_i\rangle$ now represents the original-limb of its former parent.

Fig. 2 demonstrates the effects of contract phases on an example problem instance.

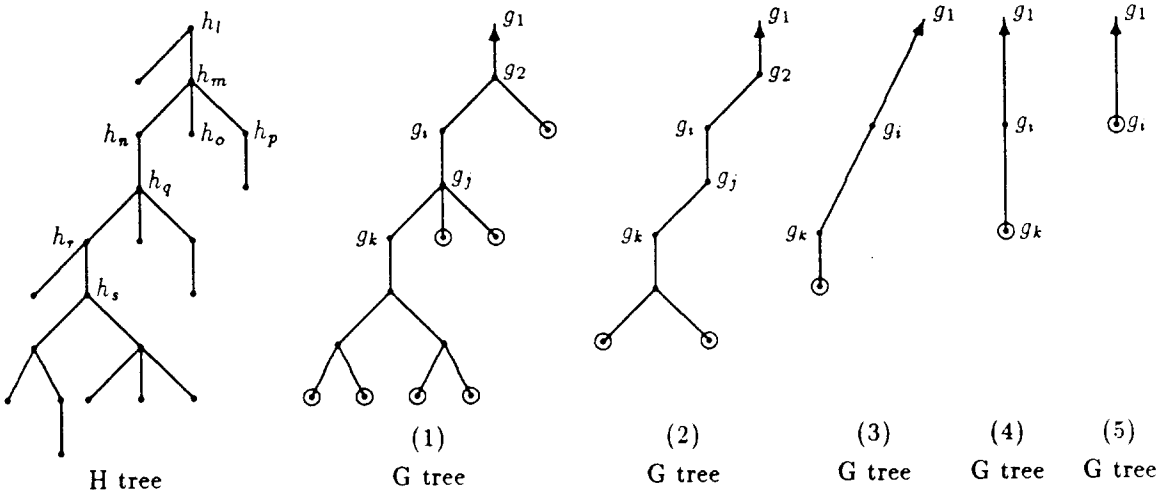
Lemma 1 *Consider a tree T rooted at a root limb. Let T be contracted to 2 nodes using dynamic tree contraction. Then the root node r of T is not involved in any rake or compress operations.*

Proof: Since T is rooted at a root limb, r has degree 1. Its one child will not become a leaf until there are exactly two nodes left in the tree. The root is never involved in a compress operation since it has no parent. □

Thus there is no need to consider the root node during contract.

Lemma 2 *Let G be a tree rooted at a root limb, and let $G\langle\star, g_i\rangle$ be a limb of G. Let* contract

An example: An instance of the subtree isomorphism problem is solved using **contract**. At each step, for a few of the G limbs, we show its original-limb and *one* of its conditionals.



phase	limb	conditional	original limb
(1)	$G\langle\star, g_i\rangle$	"$H\langle h_m, h_n\rangle$ if $H\langle h_n, h_q\rangle$"	$G\langle g_2, g_i\rangle$
	$G\langle\star, g_k\rangle$	"$H\langle h_q, h_r\rangle$ if $H\langle h_r, h_s\rangle$"	$G\langle g_j, g_k\rangle$
(2)	$G\langle\star, g_2\rangle$	"$H\langle h_p, h_m\rangle$ if $H\langle h_m, h_n\rangle$"	$G\langle g_1, g_2\rangle$
	$G\langle\star, g_i\rangle$	"$H\langle h_m, h_n\rangle$ if $H\langle h_n, h_q\rangle$"	$G\langle g_2, g_i\rangle$
	$G\langle\star, g_j\rangle$	"$H\langle h_n, h_q\rangle$ if $H\langle h_q, h_r\rangle$"	$G\langle g_i, g_j\rangle$
	$G\langle\star, g_k\rangle$	"$H\langle h_q, h_r\rangle$ if $H\langle h_r, h_s\rangle$"	$G\langle g_j, g_k\rangle$
(3)	$G\langle\star, g_i\rangle$	"$H\langle h_p, h_m\rangle$ if $H\langle h_n, h_q\rangle$"	$G\langle g_1, g_2\rangle$
	$G\langle\star, g_k\rangle$	"$H\langle h_n, h_q\rangle$ if $H\langle h_r, h_s\rangle$"	$G\langle g_i, g_j\rangle$
(4)	$G\langle\star, g_i\rangle$	"$H\langle h_p, h_m\rangle$ if $H\langle h_n, h_q\rangle$"	$G\langle g_1, g_2\rangle$
	$G\langle\star, g_k\rangle$	imbeddable in $H\langle h_n, h_q\rangle$	$G\langle g_i, g_j\rangle$
(5)	$G\langle\star, g_i\rangle$	imbeddable in $H\langle h_p, h_m\rangle$	$G\langle g_1, g_2\rangle$

Figure 2: Given two unrooted trees G and H, G is rooted at g_1 and then contracted using **contract**. A circle around a node indicates that the value of its associated limb has been computed. We show some of the partial-values and values computed at each **contract** phase. For each limb listed, we show only *one* of its many conditionals. After phase 5, we conclude that the rooted G is imbeddable in the limb $H\langle h_p, h_m\rangle$ of H. In addition, **contract** would determine that the rooted G is also limb imbeddable in $H\langle h_l, h_m\rangle$ and $H\langle h_o, h_m\rangle$.

be applied to G until the tree is contracted to 1 (final) limb. Then during the course of contracting G, the number of child limbs of $G\langle\star, g_i\rangle$ is monotonically nonincreasing, and at some phase, $G\langle\star, g_i\rangle$ has 0 or 1 nonleaf child limbs.

Proof: **Contract** only adds a new child limb to a limb during a compress operation, where it simply replaces one child limb by another. As for the second claim, observe that a limb is deleted only if it has 0 or 1 nonleaf child limbs. If $G\langle\star, g_i\rangle$ is the final limb, it has 0 children, and if not, then it gets deleted. □

Lemma 3 *Let G be a tree rooted at a root limb. Let $G\langle\star, g_i\rangle$ be a limb in G that has c child limbs. Let* **contract** *be applied to G until the tree is contracted to 1 limb. Then (1) there are no bipartite matching problems solved for node g_i if $c \leq 1$, (2) there is exactly one phase in which there are bipartite matching problems solved for node g_i if $c > 1$, and (3) prior to the matching problems during this phase, g_i will neither be deleted nor change parent node.*

Proof: There are three cases to consider. (i) Suppose $c \leq 1$. By lemma 2, it will continue to have ≤ 1 child limb. $G\langle\star, g_i\rangle$ can never be a type (2a) or type (3) limb, and thus there will not be any bipartite matching problems solved for g_i. (ii) Suppose $c > 1$ and $G\langle\star, g_i\rangle$ has ≤ 1 nonleaf limb. Then $G\langle\star, g_i\rangle$ is type (2a) or type (3b) and hence bipartite matching problems will be solved for node g_i this phase. After this phase, $G\langle\star, g_i\rangle$ has at most 1 child. It follows from the previous case that there will not be any more bipartite matching problems solved for node g_i. (iii) Suppose $c > 1$ and $G\langle\star, g_i\rangle$ has > 1 nonleaf limb. As long as $G\langle\star, g_i\rangle$ has > 1 nonleaf child limb (type (3a) or (4a)), it neither gets deleted nor changes parent nor participates in any matching problems. If one of its children is deleted as a result of a compress operation, then that child is not a leaf, and another nonleaf limb takes its place. Thus the number of children decreases only as a result of a rake operation, i.e. when a child is a leaf. By lemma 2, at some phase $G\langle\star, g_i\rangle$ goes from > 1 to ≤ 1 nonleaf child limbs. Since only leaves are deleted, the former nonleaves must now be leaves, and the node has > 1 children total. It follows from the previous case that bipartite matching problems will be solved for node g_i only this phase. □

Lemma 4 *Let G be a tree rooted at a root limb. Let* **contract** *be applied to G until the tree is contracted to 1 limb. Then during the course of contracting G, the root node of the original-limb of any limb $G\langle\star, g_i\rangle$ still in the tree is the current parent of g_i in the tree.*

Proof: The proof is by induction on the number of compress operations. The claim holds initially and is not affected by a rake operation. Assume it holds for k compress operations. Consider a limb $G\langle\star, g_i\rangle$ that combines partial-values with its parent $G\langle\star, g_j\rangle$ as a result of the $(k+1)^{\text{th}}$ compress operation. Let $G\langle g_p, g_q\rangle$ be the current original-limb of $G\langle\star, g_j\rangle$. By the induction hypothesis, g_p is the parent of g_j in the tree prior to this compress operation. As a result of this compress operation, g_p becomes the parent of g_i and $G\langle\star, g_i\rangle$ assumes the role of $G\langle\star, g_j\rangle$ in its new conditional-sets. It follows that $G\langle\star, g_i\rangle$'s new original-limb has root g_p. □

Lemma 5 *Let $G\langle\star, g'\rangle$ be a limb of a tree G after t* **contract** *phases, and let $H\langle h, h'\rangle$ be a limb of a tree H. Suppose $G\langle\star, g'\rangle$ has > 1 child limbs, which are all leaves, and that g is the parent of g' in the contracted tree. For each child limb of $G\langle\star, g'\rangle$ raked from g' this phase and previous phases, let S be the set of H limbs in which its original-limb can be imbedded. Then one can determine if the original-limb of $G\langle\star, g'\rangle$ is imbeddable in $H\langle h, h'\rangle$ by applying theorem 1.*

Proof: We must show that examining the child limbs in the contracted tree corresponds to examining child limbs in the original tree. Let $G\langle\star, g'_1\rangle, \ldots, G\langle\star, g'_k\rangle$ be the child limbs of $G\langle\star, g'\rangle$ raked from g' this phase and previous phases. By lemma 4 and the observation that the original-limb of a leaf limb does not change when the leaf is deleted, it follows that the original-limbs of these child limbs all have root g'. Since $G\langle\star, g'\rangle$ has > 1 leaves, then by lemma 2, g' has not been involved in a compress operation. Thus its original-limb is the same as its present limb, i.e. $G\langle g, g'\rangle$. Therefore in G, the original-limb of the parent limb is adjacent to the original-limbs of its children. To test whether $G\langle g, g'\rangle$ is imbeddable in $H\langle h, h'\rangle$, it follows that it suffices to consider only child limbs of $H\langle h, h'\rangle$. Thus one can apply theorem 1 to determine if $G\langle g, g'\rangle$ is imbeddable in $H\langle h, h'\rangle$. □

Lemma 6 *A* **contract** *phase as defined above preserves the invariants I above.*

Proof: Assume the invariants I are true before the **contract** phase, and consider a limb of each type. If the limb is a leaf, the **contract** phase deletes it and saves its value for the ≤ 1 phase of bipartite matching problems solved for its parent (lemma 3). If all the limb's children are leaves, **contract** will determine the set of H limbs in which (the original-limb of) the limb can be imbedded (using theorem 1 and lemma 5) and delete the leaves, creating a new leaf that satisfies the first invariant. If some are leaves and some are not, there are two cases: (i) if it has > 1 nonleaf children, **contract** deletes only the leaves, creating a node that satisfies the third invariant, and (ii) if it has exactly 1 nonleaf child, **contract** finds its conditional-set (by theorem 1 and lemma 5) and deletes the leaves, creating a node that satisfies the second invariant. If none are leaves, then the number of children at the node remains unchanged. Thus, if there were > 1 children before, **contract** leaves unchanged a node that satisfies the third invariant, and if there were exactly 1 child before, **contract** composes conditional-sets, creating a node that satisfies the second invariant. □

3.1 Pseudo-code for the algorithm

Our algorithm uses the following data structures. As in algorithm A, let **Imbed**[,] be an $(n_G - 1) \times (2n_H - 2)$ matrix, with one entry for each pair $(G\langle \star, g_i \rangle, H\langle h_j, h_k \rangle)$ of limbs associated with the rooted $G\langle g_1, g_2 \rangle$ and the nonrooted H. During the course of the algorithm, **Imbed**$[\langle \star, g_i \rangle, \langle h_j, h_k \rangle]$ will be set to 1 if and only if $G\langle \star, g_i \rangle$ is found to be limb imbeddable in $H\langle h_j, h_k \rangle$. If it is set to 1, then $G\langle \star, g_i \rangle$ must be a leaf. Thus g_i's parent will no longer change, and **Imbed**$[\langle \star, g_i \rangle, \langle h_j, h_k \rangle]$ will be valid for the rest of the algorithm. Let **Conditionals**[,,] be an auxiliary $(n_G - 1) \times (2n_H - 2) \times (2n_H - 2)$ matrix, used for storing the conditional-sets of all pairs of limbs. **Conditionals**$[\langle \star, g_i \rangle, \langle h_j, h_k \rangle, \langle h_l, h_m \rangle]$ will be set if and only if $G\langle \star, g_i \rangle$ is imbeddable in $H\langle h_j, h_k \rangle$ on the (as yet unresolved) condition that the remaining child limb of $G\langle \star, g_i \rangle$ is imbeddable in $H\langle h_l, h_m \rangle$. Recall that a short hand notation for this situation is "$H\langle h_j, h_k \rangle$ if $H\langle h_l, h_m \rangle$". Further implementation details will be described in section 3.2.

Algorithm C below gives a pseudo-code description of our subtree isomorphism algorithm.

Theorem 5 *Given two trees G and H, algorithm C determines if there is a subtree of H isomorphic to G.*

Proof: We will sketch a proof based on induction on the number of **contract** phases. After step C2, the value of any leaf limb and the partial-value of any limb with exactly one child limb has been computed. Thus the invariants I hold. By lemma 6, each **contract** phase (steps C4–C17) maintains the invariants I. By theorem 3, the WHILE loop of step C3 will succeed in contracting the tree to 1 limb, call it $G\langle \star, g_i \rangle$. By lemma 1, the two nodes remaining in G will be g_1 and g_i. Thus by lemma 4, the limb $G\langle \star, g_i \rangle$ represents the entire tree G, i.e. its original-limb is $G\langle g_1, g_2 \rangle$. Since $G\langle \star, g_i \rangle$ is now a leaf limb, it is known in which H limbs its original-limb can be imbedded. Therefore, there exists a subtree of H isomorphic to G if and only if there exists a limb $H\langle h_k, h_l \rangle$ such that **Imbed**$[\langle \star, g_i \rangle, \langle h_k, h_l \rangle] = 1$. □

3.2 Implementation details and analysis

In implementing algorithm C on a PRAM, it is helpful to preprocess the two trees after step C1. Even though G will change, the algorithm will refer to the structure of the original tree when it solves bipartite matching problems for any of its nodes. For H, use a $(2n_H - 2) \times (2n_H - 2)$ bit matrix initialized to all zeroes. For each pair of H limbs $H\langle h_i, h_j \rangle$ and $H\langle h_k, h_l \rangle$, set the bit at row $\langle h_i, h_j \rangle$, column $\langle h_k, h_l \rangle$, if and only if $H\langle h_k, h_l \rangle$ is a child limb of $H\langle h_i, h_j \rangle$, i.e. $j = k$. Using a parallel prefix algorithm[LF80], compute the index of each child among its siblings and among all children, thereby obtaining an allocation of processors which can be used throughout the algorithm. Note that there are $O(n_H^2)$ children in all, since H is not rooted and thus a node of degree k contributes $k(k-1)$ children to the total.

We can preprocess G in the same way (since G is rooted, more efficient ways do exist). To keep track of all leaves raked from a node, the algorithm will maintain a list of the child nodes

Algorithm C: Given a tree H on n_H vertices $\{h_1, \ldots, h_{n_H}\}$ and a tree G on n_G vertices $\{g_1, \ldots, g_{n_G}\}$, this algorithm determines if there is a subtree of H isomorphic to G.

C1. Select a root limb $G\langle g_1\ g_2\rangle$ of G, and for all g_i except g_1,
let $g_i.parent$ be the parent of g_i in the resulting rooted tree.

C2. Initialize the **Imbed** and **Conditionals** matrices to all zeroes. Then for each leaf limb $G\langle \star, g_i\rangle$, set **Imbed**$[\langle \star, g_i\rangle, \langle h_j, h_k\rangle]$ to 1 for all H limbs $H\langle h_j, h_k\rangle$. For each limb $G\langle \star, g_i\rangle$ with exactly one child: for all H limbs $H\langle h_j, h_k\rangle$, set **Conditionals**$[\langle \star, g_i\rangle, \langle h_j, h_k\rangle, \langle h_k, h_l\rangle]$ to 1 for all its child limbs $H\langle h_k, h_l\rangle$.

C3. WHILE there exist > 1 limbs in G DO:
C4. IN PARALLEL for all limbs $G\langle \star, g_i\rangle$ in G DO:

/* rake all leaves, update their parents accordingly */
C5. IF leaf limb
C6. delete g_i from G at the end of this phase

C7. ELSE IF all children are leaves
C8. IF node has > 1 child
C9. **Find_Imbeddings_For_Node**(g_i)

ELSE /* node has exactly 1 child */
C10. determine the set of H limbs in which limb $G\langle \star, g_i\rangle$ is imbeddable from the conditionals and the H limbs in which the remaining child $G\langle \star, g_j\rangle$ is now known to be imbeddable, e.g. if **Conditionals**$[\langle \star, g_i\rangle, \langle h_k, h_l\rangle, \langle h_m, h_n\rangle] = 1$ and **Imbed**$[\langle \star, g_j\rangle, \langle h_m, h_n\rangle] = 1$, set **Imbed**$[\langle \star, g_i\rangle, \langle h_k, h_l\rangle]$ to 1.

C11. ELSE IF some children are leaves, some are not
C12. IF node has exactly 1 nonleaf child $G\langle \star, g_j\rangle$
C13. **Find_Conditional_Imbeddings_For_Node**(g_i, g_j)

/* compress all chains */
ELSE /* no children are leaves */
C14. IF node has exactly 1 child AND node is of even parity on its chain
C15. compose the conditionals associated with g_i and $g_i.parent$, e.g. "$H\langle h_i, h_j\rangle$ if $H\langle h_k, h_l\rangle$" in parent and "$H\langle h_k, h_l\rangle$ if $H\langle h_m, h_n\rangle$" in g_i results in "$H\langle h_i, h_j\rangle$ if $H\langle h_m, h_n\rangle$". marking **Conditionals** accordingly.

C16. delete $g_i.parent$ from G at end of this phase
C17. $g_i.parent \leftarrow (g_i.parent).parent$
END
END

C18. Let $G\langle \star, g_i\rangle$ be the remaining limb in G. If there is an $H\langle h_k, h_l\rangle$ such that **Imbed**$[\langle \star, g_i\rangle, \langle h_k, h_l\rangle] = 1$, then there is a subtree of H isomorphic to G. If not, then no such subtree exists.

PROCEDURE Find_Imbeddings_For_Node(g'):

C19. IN PARALLEL for each limb $H\langle h, h'\rangle$ of H DO:
C20. IF h' has at least as many children as g' in the original trees
C21. Set up a bipartite matching problem P where the boys are the child limbs $G\langle\star, g'_1\rangle, \ldots, G\langle\star, g'_k\rangle$ of g', and the girls are the child limbs $H\langle h', h'_1\rangle, \ldots, H\langle h', h'_l\rangle$ of h'. There is an edge between boy $G\langle\star, g'_i\rangle$ and girl $H\langle h', h'_j\rangle$ if and only if $G\langle\star, g'_i\rangle$ can be limb imbedded in $H\langle h', h'_j\rangle$, i.e. **Imbed**$[\langle\star, g'_i\rangle, \langle h', h'_j\rangle] = 1$. Also, pad the boys with dummy limbs, which have edges to all the girls, to make the number of boys equal the number of girls.

C22. Find a perfect matching in P. If one exists, set **Imbed**$[\langle\star, g'\rangle, \langle h, h'\rangle]$ to 1.
END

PROCEDURE Find_Conditional_Imbeddings_For_Node(g', g'_x):

C23. IN PARALLEL for each limb $H\langle h, h'\rangle$ of H DO:
C24. IF h' has at least as many children as g' in the original trees
C25. IN PARALLEL for all child limbs $H\langle h', h'_j\rangle$ DO:
C26. Set up a bipartite matching problem P' as above, except exclude child limbs $G\langle\star, g'_x\rangle$ and $H\langle h', h'_j\rangle$ from the matching problem.

C27. Find a perfect matching in P'. If one exists, set **Conditionals**$[\langle\star, g'\rangle, \langle h, h'\rangle, \langle h', h'_j\rangle]$ to 1.
END
END

for each node. Let g_k be the i^{th} child of g_j, and let g_l be the only remaining child of g_k. If g_l deletes g_k as part of a compress operation, then g_l replaces g_k as the i^{th} child of g_j. It follows from lemma 3 that this approach retains a list of the leaves raked from a node up until the one phase in which bipartite matching problems are solved for the node.

Here is a step by step analysis of algorithm C. Recall that $M(n) = n^{2+\epsilon}$ is the best sequential arithmetic time bound for multiplying two $n \times n$ matrices (ϵ is < 0.4). It is convenient to describe the implementation of some steps using concurrent write instructions. In all cases, these instructions will be simulated by an equivalent set of exclusive write instructions.

- For step C1, we root the guest tree G since we can assume it is no bigger than H. Given a list of the edges of G, one can find a degree 1 node using concurrent write (arbitrary model) in $O(1)$ time with n_G processors. Then root the tree using the Euler tour technique for trees[TV85], in $O(\log n_G)$ time and $n_G/\log n_G$ processors. The preprocessing of G and H that follows requires $O(\log n_H)$ time and $n_H^2/\log n_H$ processors. Given this preprocessing, step C2 takes $O(1)$ time and $n_G n_H^2$ processors.

- Steps C4–C17 perform one **contract** phase. The tests in steps C3, C5, C7, C8, C11, C12, and C14 depend on the structure of the current tree. In each case, we wish to determine if a node has 0, 1, or > 1 children of a particular type. The most time-efficient way to perform these tests is using concurrent write (arbitrary model). Each node g_i with parent g_j writes i in cell j, then reads cell j to see if it has succeeded in its write attempt. If not, it complains to its parent. This takes $O(1)$ time and n_G processors. Alternatively, the algorithm could preprocess the tree each time, at a cost still less than that of other steps inside this WHILE loop. The second test in step C14 can be done in $O(\log n_G)$ time with n_G processors: the index of each node in a chain is computed by $O(\log n_G)$ applications of pointer jumping.

- Steps C6, C16, and C17 can be done in $O(1)$ time and n_G processors.

- Step C10 takes $O(1)$ time and n_H^2 processors using concurrent write for each g_i. For step C15, perform a boolean matrix multiplication for each g_i in $O(1)$ time and $M(n_H)$ processors using concurrent write. Note that a temporary matrix is needed here, since **Conditionals** is updated in place.

- For step C9, i.e. steps C19–C22, for each g_i, $O(n_H)$ bipartite matching problems of size $\leq n_H$ are solved in parallel. Using algorithm B, step C22 takes $O(\log^2 n_H)$ time and $n_H^3 M(n_H) \log\log n_H / \log n_H$ processors for all matchings for each g_i. Steps C20 and C21 use the preprocessing information obtained for H and obtained and maintained for G to set up the adjacency matrices for the matching problems.

- Similarly, for step C13, i.e. steps C23–C27, for each g_i, $O(n_H^2)$ bipartite matching problems of size $\leq n_H$ are solved in parallel. Thus step C27 takes $O(\log^2 n_H)$ time and $n_H^4 M(n_H) \log\log n_H / \log n_H$ processors for all matchings for each g_i.

- Step C18 can be done in $O(1)$ time and n_H processors using concurrent write.

By theorem 3, there will be $O(\log n_G)$ iterations of the WHILE loop, and so steps C3–C17 will take $O(\log n_G \log^2 n_H)$ time. Thus any instruction above using concurrent write can be simulated by $O(\log n)$ equivalent exclusive write instructions at no increase to the running time or processor count. By lemma 3, step C9 or step C13 will be executed at most once for each g_i. It follows that algorithm C runs in $O(\log n_G \log^2 n_H)$ time on a CREW PRAM with $n_G n_H^4 M(n_H) \log\log n_H / \log n_G \log n_H$ processors, i.e. $o(n_G n_H^{6.4})$ processors. In the next section, we show how the processor count can be significantly reduced.

In order to exhibit the mapping between G and a subtree of H, we will add new instructions to algorithm C as follows. We save sufficient information about each deleted node to enable us to uniquely determine a mapping while expanding the tree top down, starting with the final limb. While contracting the tree, count the number of **contract** phases applied so far, in order to save the "time" each node was deleted. When a node is deleted as a result of a rake operation, also

save the name of its parent; for a compress operation, save the name of its child. The precise instructions added to algorithm C are listed below (shown properly indented to fit into algorithm C). **Save_Imbed** is an $(n_G - 1) \times (2n_H - 2)$ matrix and **Save_Conditionals** is an $(n_G - 1) \times (2n_H - 2) \times (2n_H - 2)$ matrix.

Given the above additions to algorithm C, the procedure **Expand_Tree** shown below can be used to exhibit the mapping. Initially, we know that a mapping exists with the final limb $G\langle \star, g_i \rangle$ imbedded in some limb $H\langle h_k, h_l \rangle$. Let $g_i.home = H\langle h_k, h_l \rangle$. Let p be the total number of **contract** phases used to contract G. **Expand_Tree** performs a series of **expand** phases, with the i^{th} such phase splicing back into the tree all nodes and edges deleted at the $(p - i + 1)^{\text{th}}$ **contract** phase, while maintaining the "home" in H for each limb in the tree. Prior to the final **expand** phase, these "home" limbs will typically be scattered throughout H, with the missing edges filled in by subsequent **expand** phases.

Clearly the time and processor count for this procedure is bounded by the time and processor count for algorithm C.

4 Processor efficiency

We have recast the subtree isomorphism problem as a problem on limbs, as in Matula's algorithm, in order to save having to try out all possible roots for the trees. In this section, we show how to reduce further the number of processors needed. First, we observe that the algorithm need not *construct* any matchings in order to determine a home H limb for the rooted G. Algorithm C determines whether a particular pair of limbs can be matched in a perfect matching by solving a matching problem with the pair removed from the graph. Steps C22 and C27 simply test if a perfect matching exists. We can save processors by using an algorithm for *deciding* whether a perfect matching exists while contracting G, and an algorithm for *constructing* the matching while expanding G. (There are certain advantages to constructing the matchings as we contract G: see section 6).

We modify our algorithm as follows. First, use a *decision* algorithm for steps C22 and C27. Step C22a is removed and step C27a will simply save the name of the remaining child. Then, determine the mapping while expanding the tree. We will maintain the invariant that the *home* H limb is known for every limb in the current G tree. Initially, the home limb is known for the final limb in the contracted G tree. As before, determine the home of each new G limb as it is added back to its tree. We modify **Expand_Tree** as follows. (i) If g_i was raked for a matching problem (C31), and no matching problem has been solved for $G\langle \star, g_i \rangle$, construct the matching and save the results in **Save_Imbed**. In this case, the home limb $H\langle h_j, h_k \rangle$ of the parent limb $G\langle \star, g_i \rangle$ is known, so it suffices to solve a matching problem between the children of $G\langle \star, g_i \rangle$ and the children of $H\langle h_j, h_k \rangle$. (ii) If g_i was raked for a conditional matching problem (C35), and no matching problem has been solved for $G\langle \star, g_i \rangle$, construct the matching and save the results in **Save_Conditionals**. In this case, both the home limb $H\langle h_k, h_l \rangle$ for $G\langle \star, g_i \rangle$ and the home limb $H\langle h_l, h_m \rangle$ for the remaining child $G\langle \star, g_j \rangle$ of $G\langle \star, g_i \rangle$ are known, so it suffices to solve a matching problem between the children of $G\langle \star, g_i \rangle$ (except for $G\langle \star, g_j \rangle$) and the children of $H\langle h_k, h_l \rangle$ (except for $H\langle h_l, h_m \rangle$).

The running time for expanding the tree is $O(\log n_G \log^2 n_H)$, using algorithm B for constructing the perfect matchings. The processor savings can be seen from the following lemma.

Lemma 7 *While expanding the tree, there is at most one bipartite matching problem solved for each node in the rooted H.*

Proof: An H limb can not be host to two limbs of G. Since H is rooted, each node is associated with a unique limb in H. By lemma 3, there was at most one phase of bipartite matching problems solved for a node g_i while G was contracted. Thus, in the modified **Expand_Tree** procedure, there will be at most one bipartite matching problem solved for g_i while G is expanded. Let $H\langle h, h' \rangle$ be the home limb for $G\langle \star, g_i \rangle$ at the time t that a matching problem is solved for g_i. Once assigned, a home limb does not change unless the node's parent changes. But by lemma 3, g_i's parent did not change at any time $< t$, and thus $H\langle h, h' \rangle$ will continue to be the home limb for $G\langle \star, g_i \rangle$ as the tree is expanded. It follows that there is only one matching problem solved for h'. □

Algorithm C (revisions): These instructions, when added to algorithm C, can be used with the procedure Expand_Tree in order to exhibit the isomorphic mapping between the two trees.

C6a. save for g_i the current "time" and the name of its parent (g_i.parent)

C10a. **Save_Imbed**$[\langle \star, g_j \rangle, \langle h_k, h_l \rangle] \leftarrow \langle h_m, h_n \rangle$

C15a. **Save_Conditionals**$[\langle \star, g_p \rangle, \langle h_i, h_j \rangle, \langle h_m, h_n \rangle] \leftarrow \langle h_k, h_l \rangle$,
where g_p is the parent of g_i

C16a. save for g_i.parent the current "time", the name of
its child (g_i), and the name of its remaining grandchild

C22a. IF perfect matching M exists, save it. In particular, for each child limb
$G\langle \star, g_i' \rangle$ of $G\langle \star, g' \rangle$, **Save_Imbed**$[\langle \star, g_i' \rangle, \langle h, h' \rangle] \leftarrow \langle h', h_j' \rangle$,
where $G\langle \star, g_i' \rangle$ is matched with $H\langle h', h_j' \rangle$ in M.

C27a. IF perfect matching M exists, save it and the remaining child.
In particular, for each child limb $G\langle \star, g_i' \rangle$ of $G\langle \star, g' \rangle$ (except $G\langle \star, g_x' \rangle$),
Save_Conditionals$[\langle \star, g_i' \rangle, \langle h, h' \rangle, \langle h', h_j' \rangle] \leftarrow \langle h', h_k' \rangle$, where
$G\langle \star, g_i' \rangle$ is matched with $H\langle h', h_k' \rangle$ in M, and g_i'.remaining_child $\leftarrow g_x'$.

PROCEDURE Expand_Tree:

C28. Let t step by -1 from the number of **contract** phases down to 1
C29. IN PARALLEL for limbs $G\langle \star, g_i \rangle$ in G DO:
C30. IF g_i deleted at time t
C31. IF raked for a matching problem (i.e. by step **C9**)
C32. OR rake of last child (i.e. by **C10**)
C33. add g_i to tree
C34. g_i.home $\leftarrow$ **Save_Imbed**$[\langle \star, g_i \rangle, \langle h_k, h_l \rangle]$, where
$H\langle h_k, h_l \rangle$ is (g_i.parent).home

C35. ELSE IF raked for a conditional matching problem (i.e. by **C13**)
C36. add g_i to tree
C37. g_i.home $\leftarrow$ **Save_Conditionals**$[\langle \star, g_i \rangle, \langle h, h' \rangle, \langle h', h_j' \rangle]$,
where $H\langle h, h' \rangle$ is (g_i.parent).home
and $H\langle h', h_j' \rangle$ is (g_i.remaining_child).home

ELSE /* deleted by a compress operation (i.e. by **C16**) */
C38. g_i.parent $\leftarrow$ (g_i.child).parent /* splice into tree */
C39. (g_i.child).parent $\leftarrow g_i$
C40. g_i.home $\leftarrow$ (g_i.child).home
C41. (g_i.child).home $\leftarrow$ **Save_Conditionals**$[\langle \star, g_i \rangle, \langle h_j, h_k \rangle, \langle h_l, h_m \rangle]$,
where $H\langle h_j, h_k \rangle$ is g_i's new home
and $H\langle h_l, h_m \rangle$ is (g_i.grandchild).home

Define the work of an algorithm to be the sum over all processors p_i of the number of PRAM instructions/operations executed by p_i during the course of the algorithm. Let w be the amount of work to expand the tree using the modified **Expand_Tree** procedure. Clearly w is dominated by the work to construct the matchings. Let d_j be the degree of node h_j. By lemma 7,

$$\begin{aligned} w &\leq \sum_{j=1}^{n_H} d_j^2 M(d_j) \log d_j \log\log d_j \\ &\leq n_H M(n_H) \log n_H \log\log n_H \sum_{j=1}^{n_H} d_j \\ &\in O(n_H^2 M(n_H) \log n_H \log\log n_H) \end{aligned}$$

(This bound holds even if the word size of the PRAM is 1 bit).

We will now analyze the complexity of contracting the guest tree using a *decision* algorithm for bipartite matching instead of a *search* algorithm (which uses more processors). Borodin, von zur Gathen, Hopcroft's algorithm[BvzGH82] for deciding if a bipartite graph has a perfect matching assigns random integer weights of only $O(\log n)$ bits to each of the edges in the graph, and runs in time $O(\log^2 n)$. A processor-efficient version of their algorithm computes determinants over Zp, the integers modulo some suitable prime p of size $O(n^4)$[RV84]. This can be done on a PRAM using Preparata and Sarwate's matrix inversion algorithm[PS78] with $O(\sqrt{n}M(n))$ work, since all operations involve $O(\log n)$-bit numbers. (Galil and Pan[GP85b] have a slightly better algorithm for inverting matrices over Z_p). Since algorithm C solves $O(n_G n_H^2)$ bipartite matching problems, and the work for its other steps is $O(n_G M(n_H))$, it follows that the work to contract the tree is $O(n_G n_H^{2.5} M(n_H))$.

The tree can be contracted with less work by observing that the inverse of a matrix can be used to decide perfect matching problems on its minors. The $a_{i,j}$ entry of the adjoint of A contains the determinant of $A_{j,i}$ (the $(j,i)^{\text{th}}$ *cofactor*). By testing whether a cofactor is 0, we can determine if a perfect matching exists when the parent limb and any one H limb are left out of the matching. From Rabin and Vazirani[RV84], it follows that this holds even when the adjoint is computed over Z_p. In order to find in which parent limbs $H\langle h_j, h_k\rangle$ adjacent to h_k the limb $G\langle \star, g_i\rangle$ can be imbedded, solve one bipartite matching problem where the boys are the child limbs of $G\langle \star, g_i\rangle$, and the girls are the child limbs $H\langle h_k, h_j\rangle$ adjacent to h_k. Pad the boys to match the number of girls as before, except that one of these dummy boys is designated to correspond to $G\langle \star, g_i\rangle$. From the matrix inverse computed, test the cofactors of the dummy row corresponding to the parent $G\langle \star, g_i\rangle$. $G\langle \star, g_i\rangle$ is imbeddable in $H\langle h_j, h_k\rangle$ if and only if its cofactor is $\neq 0$. Thus for each node g_i, $\leq n_H$ bipartite matching problems (one per each h_j) will be solved, each with $\leq d_j = \deg(h_j)$ children. Let w be the work to solve these matching problems. Then

$$\begin{aligned} w &\leq n_G \sum_{j=1}^{n_H} \sqrt{d_j} M(d_j) \\ &\in O(n_G \sqrt{n_H} M(n_H)) \end{aligned}$$

This technique can be applied to conditional matching problems as well, where the parent is left out of the graph and the dummy row corresponds to the remaining child (or vice-versa). For each node g_i, $\leq 2n_H - 2$ bipartite matching problems are solved, each with $\leq n_H$ children. It follows that the work for contracting the tree is $O(n_G n_H^{1.5} M(n_H))$.

Given the analysis of the work for contracting and expanding the tree, we can apply Brent's scheduling lemma[Bre74] to determine the number of processors needed. Before each phase t, the algorithm can determine the operations to be done during the phase and allocate the processors accordingly in time $O(\log n_H)$ using $n_G n_H^2$ processors. Since there are $O(\log n_G)$ phases, this overhead increases the running time and work by at most a constant factor. Let **algorithm C'** be the improved version of algorithm C which uses the above steps to save processors and to print out the mapping. Then the following theorem follows from theorem 5.

Theorem 6 *Given a tree H on n_H vertices and a tree G on n_G vertices, algorithm C' determines, with probability $\geq 1/2$, if there is a subtree of H isomorphic to G, and exhibits the mapping. It runs in time $t \in O(\log n_G \log^2 n_H)$ on a CREW PRAM with $(n_G + n_H^5 \log n_H \log\log n_H) n_H^{1.5} M(n_H)/t$ processors, i.e. $O(\log^3 n)$ time with $n^{2.5} M(n)/\log^3 n$ processors.*

We omit the proof.

5 Reducing matching to subtree isomorphism

In this section we show that perfect bipartite matching is log space reducible to subtree isomorphism. Let $G = (X, Y, E)$ be a bipartite graph, where $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$. We will construct trees T_x, T_y corresponding to the vertex sets X and Y, such that every imbedding of T_x in T_y yields, in a natural way, a perfect matching in G. It is convenient to view T_x and T_y as rooted at R_x and R_y respectively. This creates no obstacle since our construction forces R_x to be mapped to R_y in any imbedding. The structure of the trees is as follows:

T_x : R_x has $n+2$ children - $X_1, X_2, \ldots, X_n, V_1, V_2$. X_i corresponds to vertex x_i in G. V_1 and V_2 have no children. For $1 \leq i \leq n$, X_i is the parent of i children, $X_{i,j}$, each of which is a root of a path of length $n - i + 1$.

T_y : R_y has $n+2$ children - $Y_1, Y_2, \ldots, Y_n, U_1, U_2$. Y_i corresponds to vertex y_i in G. V_1 and V_2 have no children. For $1 \leq i \leq n$, Y_i is the parent of n children, $Y_{i,j}$, where $Y_{i,j}$ is the root of a path of length $n - j + 1$ if $\{y_i, x_j\} \in E$ and length $n - j$ otherwise.

Note that this reduction can clearly be performed in log space.

Lemma 8 *The subtree rooted at X_i can be imbedded in the subtree rooted at Y_j if and only if $\{x_i, y_j\} \in E$.*

Proof: By construction, the trees rooted at $Y_{j,1}, \ldots, Y_{j,i-1}$ are paths of length at least $n-i+1$, and the trees rooted at $Y_{j,i+1}, \ldots, Y_{j,n}$ are paths of length less than $n - i + 1$. Furthermore, the tree rooted at $Y_{j,i}$ has length at least $n - i + 1$ if and only if $\{x_i, y_j\} \in E$. Now, since the children of X_i are roots of paths of length $n - i + 1$ and there are i of them, the claim follows. □

Lemma 9 *In any imbedding of T_x in T_y, R_x is mapped to R_y.*

Proof: The degrees of R_x and R_y are $n + 2$. All the other vertices in T_y have smaller degree. □

Theorem 7 *T_x is imbeddable in T_y if and only if G has a perfect matching.*

Proof: Let $M = \{\{x_1, y_{(1)}\}, \ldots, \{x_n, y_{(n)}\}\}$ be a perfect matching of G. By lemma 8, the subtree rooted at X_i is imbeddable in the subtree rooted at $Y_{(i)}$ for all i. It follows that T_x is imbeddable in T_y.

Conversely, assume there is an imbedding of T_x in T_y. By lemma 9, R_x is mapped to R_y, and it follows that X_i is mapped to some $Y_{(i)}$ for each i. By lemma 8, $\{x_i, y_{(i)}\} \in E$ for all i, and thus the set of edges $\{\{x_1, y_{(1)}\}, \ldots, \{x_n, y_{(n)}\}\}$ constitutes a perfect matching of G. □

Corollary 1 *The problem of* deciding *if a bipartite graph has a perfect matching is log space reducible to the problem of deciding if a tree is isomorphic to a subtree of another tree.*

Corollary 2 *The problem of* constructing *a perfect matching in a bipartite graph is log space reducible to the problem of constructing an imbedding of a tree into another tree.*

Lingas and Karpinski[LK87] independently discovered an NC^1 reduction of bipartite perfect matching to subtree isomorphism.

Theorem 8 *The number of imbeddings of T_x in T_y is $2n!(n-1)!(n-2)!\cdots 2!$ times the number of perfect matchings of G.*

Proof: A perfect matching, $M = \{\{x_1, y_{(1)}\}, \ldots, \{x_n, y_{(n)}\}\}$ of G induces a unique mapping of X_i's to Y_j's. The subtree rooted at X_i can be imbedded in exactly $i!$ ways into the subtree rooted at $Y_{(i)}$. The vertices V_1, V_2 can be mapped in two ways to U_1, U_2. The theorem follows. □

Corollary 3 *The problem of determining the number of imbeddings of a tree in another tree is #P-complete.*

6 Remarks

We have presented a randomized parallel algorithm for deciding whether there is a subtree of a first tree isomorphic to a second tree, and finding the mapping if it exists. Since the Mulmuley, Vazirani, Vazirani algorithm requires $O(n_H^2 M(n_H) \log n_H \log\log n_H)$ work, we have reduced the work of our algorithm down to $\max(1, n_G/\sqrt{n_H} \log n_H \log\log n_H)$ times the work to solve *one* matching problem (from $n_G n_H^2$

times). While our algorithm is not optimal when compared with the sequential algorithm, this is a positive indication that most of the loss was already present in the bipartite matching algorithm. Nevertheless, it would be nice to shave the extra $o(\sqrt{n})$ off the work. Matula has demonstrated, for the sequential case, that subtree isomorphism can be done in the same number of operations as bipartite matching. It is an open question whether the same can be said for the parallel case.

There are two additional observations that Matula makes in order to reduce the work (i.e. time) for the sequential algorithm. First, Matula exploits the fact that the work for sequential bipartite matching[HK73] is proportional to both the number of boys p and the number of girls q (i.e. the work is $O(pq^{1.5}), p \leq q$). In the parallel case, the matrices must be square in order to apply the known fast bipartite matching algorithms. If there were a fast parallel decision (search) algorithm for bipartite matching which required, say, $pq^{1.9}$ ($pq^{3.4}$) processors, then the work for our algorithm would equal the work for constructing a matching. Second, Matula reduces the number of bipartite matchings performed, and thus the work, by using *breadth first search* to construct all $\deg(h_j)$ matchings involving a node h_j from a single matching involving h_j. If we were to use a search algorithm for bipartite matching when contracting the tree, we could likewise use a parallel shortest path algorithm (which uses matrix multiplications) to construct all the $(\deg(h_j))^2$ conditional matchings involving h_j from a single matching involving h_j. Such an implementation of our algorithm runs in $O(\log n_G \log^2 n_H)$ time on a CREW PRAM with $n_G n_H^2 M(n_H) \log\log n_H / (\log n_G \log n_H)$ processors. However, since we use a decision algorithm for bipartite matching when contracting the tree (which gives us less overall work), we do not construct any first matching from which the others could be constructed. It is an open question how to extract the desired $\deg(h_j)$ cofactors of each of the $\deg(h_j)$ appropriate minors from a single adjoint matrix, without increasing the time and processor bounds present in the Pan or Preparata and Sarwate matrix inversion algorithms.

The bipartite matching algorithms used are both *Monte Carlo* algorithms, i.e. they run in fixed time and produce the correct answer with high probability (higher success probabilities can always be achieved by repeating the algorithms). Given a bipartite graph $G = (U, V, E)$ with $|U| = |V| = n$ nodes, and a matching M of size $< n$, we can test whether M is a maximum matching for G as follows. Let $G' = (U', V', E')$ be the graph obtained from G by directing all matched edges from U to V, and all unmatched edges from V to U. There exists an *augmenting path* in G if and only if there exists a path from some unmatched node in V' to some unmatched node in U'. The latter can be determined, by taking the transitive closure of the adjacency matrix for G', in time $O(\log n)$ on a CRCW PRAM with $M(n)/\log n$ processors. (We can *construct* an augmenting path, by performing a shortest path calculation on G', in time $O(\log^2 n)$ on a CREW PRAM with $M(n)/\log^2 n$ processors). The Mulmuley, Vazirani, Vazirani algorithm for constructing a perfect matching can be used in parallel with the above test. The result is a *Las Vegas* algorithm: it runs in expected time $O(\log^2 n)$ on a CREW PRAM and always produces the correct answer. If we use the Mulmuley, Vazirani, Vazirani algorithm while contracting the guest tree, our Monte Carlo algorithm can similarly be extended to a Las Vegas algorithm with $n_G n_H^2 M(n_H) \log\log n_H / (\log n_G \log n_H)$ processors, and expected running time $O(\log n_G \log^2 n_H)$.

Alternatively, while contracting the guest tree, we could use the Galil and Pan processor-efficient version[GP85a] of the Karp, Upfal, Widgerson randomized algorithm[KUW86] for constructing a perfect matching. This leads to a Las Vegas algorithm using only $n_G n_H M(n_H) \log\log n_H / \log n_G$ processors, but requiring $O(\log n_G \log^3 n_H)$ expected time.

Throughout this paper, we have made the reasonable assumption that the PRAM word size is $O(\log n)$. If we consider arithmetic-PRAM's, which can perform addition, subtraction, multiplication, and division of arbitrary length numbers in one step, our algorithm uses only $n_G n_H M(n_H)/(\log n_G \log^2 n_H)$ processors. This follows from the fact that the Mulmuley, Vazirani, Vazirani algorithm requires just $M(n)$ arithmetic-PRAM processors to construct a perfect matching in a bipartite graph with n boys and n girls.

The case where G (or H) has bounded maxi-

mum degree d can be done *deterministically* in time $O(d \log^2 n_H \log n_G)$ on a CREW PRAM. Simply solve the bipartite matching problems in our algorithm using d applications of the above method for constructing an augmenting path in parallel.

With appropriate implementation, our algorithm is in RNC^3. To see this, observe (i) without the matchings, our algorithm runs in $O(\log^2 n)$ time, and can be implemented on an NC circuit of depth $O(\log^3 n)$, and (ii) the bipartite matching algorithms used are in RNC^2.

References

[Bre74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21:201–206, 1974.

[BvzGH82] Allan Borodin, Joachim von zur Gathen, and John Hopcroft. Fast parallel matrix and GCD computations. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 65–71, IEEE, October 1982.

[GP85a] Zvi Galil and Victor Pan. Improved processor bounds for algebraic and combinatorial problems in RNC. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 490–495, IEEE, October 1985.

[GP85b] Zvi Galil and Victor Pan. Improving the efficiency of parallel algorithms for the evaluation of the determinant and the inverse of a matrix. 1985. manuscript.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[KUW86] Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6(1):35–48, 1986.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *JACM*, 27(4):831–838, 1980.

[LK87] Andrzej Lingas and Marek Karpinski. Subtree isomorphism and bipartite perfect matching are mutually NC reducible. 1987. submitted for publication.

[LT75] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. In *Proceedings of the Waterloo Conference on Theoretical Computer Science*, 1975.

[Mat78] David W. Matula. Subtree isomorphism in $O(n^{5/2})$. *Annals of Discrete Mathematics*, 2:91–106, 1978.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its applications. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, IEEE, October 1985.

[MVV87] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[Pan85] Victor Pan. Fast and efficient parallel algorithms for the exact inversion of integer matrices. In S.N. Maheshwari, editor, *Lecture Notes in Computer Science, Vol. 206*, pages 504–521, Springer-Verlag, 1985. Proceedings of the 5th Foundations of Software Technology and Theoretical Computer Science Conference.

[PS78] F.P. Preparata and D.V. Sarwate. An improved parallel processor bound in fast matrix inversion. *Information Processing Letters*, 7(3):148–150, 1978.

[RV84] Michael O. Rabin and Vijay V. Vazirani. *Maximum Matchings in General Graphs through Randomization.* Technical Report TR-15-84, Aiken Computation Laboratory, Harvard University, October 1984.

[TV85] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862–874, 1985.