Discrete Applied Mathematics 29 (1990) 35–62
North-Holland

SUBTREE ISOMORPHISM IS IN RANDOM NC

Phillip B. GIBBONS and Richard M. KARP
Computer Science Division, University of California, Berkeley, CA 94720, USA

Gary L. MILLER
Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA

Danny SOROKER
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA

Received 12 May 1989

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. The number of processors required by the algorithm is polynomial in n. 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 bipartite perfect matching to subtree isomorphism. Finally, we present some techniques to reduce the number of processors used by the algorithm.

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. In Fig. 1, G_1 is isomorphic to a subtree of H, but G_2 is not. The subtree isomorphism problem has applications in the area of pattern recognition.

Subtree isomorphism is interesting theoretically since it is in P (the fastest sequential algorithm, due to Matula [15], runs in $O(n^{2.5})$ time, where n is the number of nodes in H), yet most natural generalizations are NP-complete. Examples include the case where G is a forest and H is a tree (*subforest isomorphism*) and the case where G is a directed tree and H is a directed acyclic graph [7]. Given that subtree isomorphism has an efficient sequential algorithm, it is natural to ask whether the problem has a fast parallel algorithm, i.e., is in NC. (A problem is in NC if it can be computed by a log-space uniform family of Boolean circuits of polynomial size and polylog depth [11, 19].) Miller and Reif [16] showed that the *tree isomorphism* problem, which can be viewed as the subtree isomorphism problem restricted to the case where G and H have the same number of nodes, is in NC. This paper presents

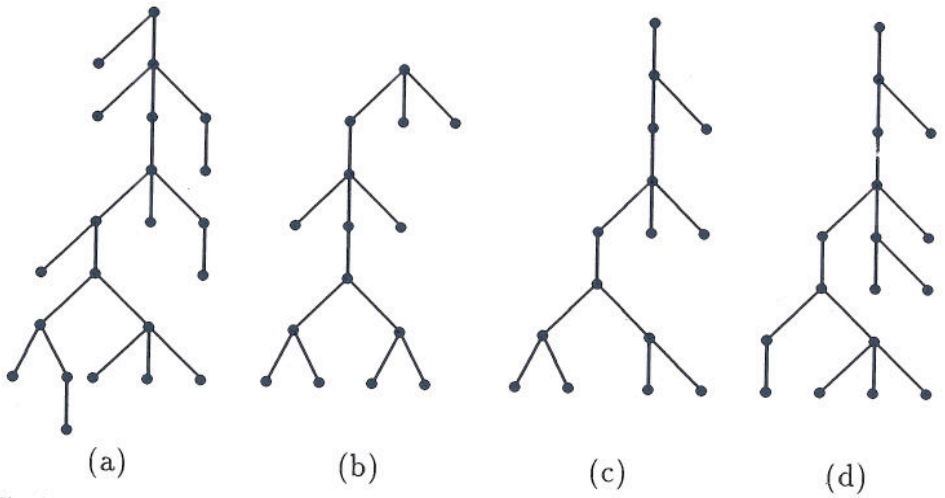


Fig. 1. (a) A host tree H. (b) A guest tree G_1, which is isomorphic to a subtree of H. (c) A tree isomorphic to G_1, which is oriented to demonstrate that G_1 is indeed isomorphic to a subtree of H. (d) A guest tree G_2 that is not isomorphic to a subtree of H.

two results on the parallel complexity of subtree isomorphism: (1) we present an $O(\log^3 n)$ time randomized parallel algorithm for the problem, and (2) we show that the parallel complexity of subtree isomorphism is closely related to the *bipartite perfect matching* problem by presenting a log-space reduction from bipartite perfect matching to subtree isomorphism. Independently, Karpinski and Lingas [13] developed an RNC^3 algorithm for subtree isomorphism and an NC^1 reduction of bipartite perfect matching to subtree isomorphism. These results show that finding an NC algorithm for subtree isomorphism is equivalent to finding an NC algorithm for bipartite perfect matching. The latter is a well-known open problem [12, 17].

Our parallel model of computation is the CREW PRAM. For a description of the PRAM model, and its relationship to the class NC, see [11]. We assume the word size of the PRAM is $c \log n$ for some constant c. Our algorithm exhibits the mapping between G and H, if such a mapping exists. With a few techniques to reduce the processor count, the algorithm uses $o(n^{5.4})$ processors, the number of processors needed for *one* bipartite matching problem on n nodes using the fastest algorithm for bipartite matching to date [17]. More precisely, let $M(n)$ be the number of bit operations required by a CREW PRAM to multiply two $n \times n$ Boolean matrices in $O(\log n)$ time. Then our algorithm uses $n^3 M(n) \log \log n / \log n$ processors.

Our parallel algorithm is based on Matula's sequential algorithm. The main obstacle to developing a fast parallel algorithm from this sequential algorithm is that its running time is proportional to the height of the guest tree. But by adapting the dynamic tree contraction technique of Miller and Reif [16], we show that subtree isomorphism is in random NC (RNC). Dynamic tree contraction 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 [2]. The subtree isomorphism algorithm of Karpinski and Lingas is based on the latter method. In both algorithms, randomization is needed (solely) to perform the bipartite matching computations.

Miller and Reif [16] use dynamic tree contraction to develop an NC algorithm for the related problem of finding canonical labels for all subtrees (*maximal subtree isomorphism*). A subtree rooted at node u in a rooted tree T is *maximal* if it contains all descendants of u in T. The problem is to assign 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 problem differs from the subtree isomorphism problem, in which the subtrees of H are not necessarily maximal.

In Section 2, we describe an algorithm for solving a rooted version of subtree isomorphism. In Section 3, our algorithm is extended to the general (unrooted) case. We present pseudo-code for the algorithm, as well as details on how to implement the algorithm on a CREW PRAM. Section 4 describes how to reduce the number of processors used, and Section 5 presents a log-space reduction of bipartite matching to subtree isomorphism. Finally, in Section 6, we present extensions of our results to other models of computation and to special cases of the subtree isomorphism problem.

2. Rooted subtree isomorphism

We first present some definitions. Let $e=(u, v)$ be a (directed) edge in a rooted tree T, where v is the parent of u. We will consider such an edge to be directed out of u and into v. For each edge f directed into u, f is a *child edge* or child of e and e is a *parent edge* or parent of f. An edge with no children is called a *leaf edge*; an

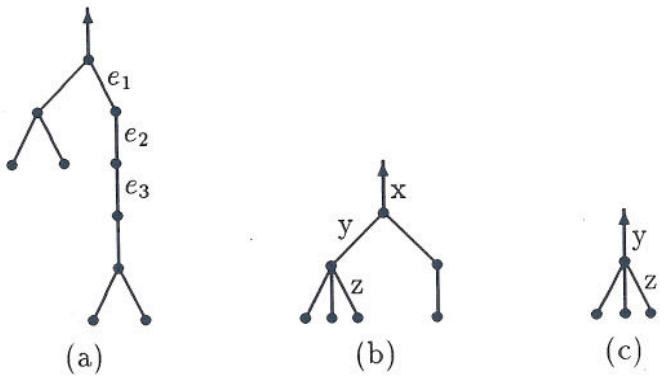


Fig. 2. (a) Rooted tree with unary chain (e_1, e_2, e_3). (b) Limb $L(x)$. (c) Limb $L(y)$, a child limb of $L(x)$.

edge with one child is called a *unary edge*. A *unary chain* in T is a maximal sequence of unary edges $e_1, e_2, \ldots, e_k$ where e_{i+1} is the child edge of e_i for $1 \leq i < k$, and e_k has exactly one child edge and that child is not a leaf edge. Edge e_i, for i odd (even), is said to be of odd (even) *parity* on its unary chain. In Fig. 2, the sequence (e_1, e_2, e_3) constitutes a unary chain. The *limb* $L(e)$ associated with the directed edge $e = (u, v)$ is the (rooted) subtree of T whose node set $V = \{u, v\} \cup \{i \mid i$ is a descendant of u in $T\}$, and whose edge set $E = \{(x, y) \mid (x, y)$ is an edge in T and $x \in V, y \in V\}$. Each parent (child) edge of e defines a *parent limb* (*child limb*) of $L(e)$. Each leaf edge defines a *leaf limb*. The *height* of a limb is the number of edges in its longest root-to-leaf path. The *level* of a limb $L(e)$ in a limb $L(f)$ is the number of edges in the path from e to f, inclusive. In Fig. 2, for example, $L(y)$ is a child limb of $L(x)$ of height two and $L(z)$ is a leaf limb in $L(x)$ of level three. A *limb-rooted* tree is a rooted tree with exactly one edge directed into the root node. Given two limbs $L(e)$ and $L(f)$, we say that $L(e)$ is *imbeddable* in $L(f)$ (equivalently, $L(f)$ is a *home* for $L(e)$) if and only if there exists an isomorphic mapping from $L(e)$ to a subtree of $L(f)$ such that e is mapped into f.

Let $y = (a, b)$ be an edge in a limb $L(x)$. The *partial limb* $L(x) - L(y)$ is the (rooted) subtree of $L(x)$ obtained by deleting all of $L(y)$, except for node b, from $L(x)$ (see Fig. 3). Given two partial limbs $L(i) - L(j)$ and $L(x) - L(y)$, we say that $L(i) - L(j)$ is a *home* for $L(x) - L(y)$ if and only if the level of $L(j)$ in $L(i)$ is the same as the level of $L(y)$ in $L(x)$ and there exists an isomorphic mapping from $L(x) - L(y)$ to a subtree of $L(i) - L(j)$ such that x is mapped to i. In Fig. 3, for example, $L(i) - L(j)$ is a home for $L(x) - L(y)$.

2.1. *Developing a fast parallel algorithm*

Matula's sequential algorithm for subtree isomorphism makes use of the following procedure:

Procedure *P*. Let $L(g)$ and $L(h)$ be a guest limb and a host limb, respectively, for which we already know the following: for each child edge x of g and each child edge

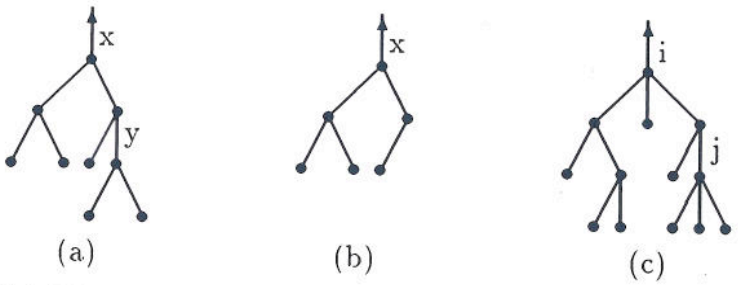


Fig. 3. (a) Limb $L(x)$. (b) Partial limb $L(x) - L(y)$. (c) Limb $L(i)$. Partial limb $L(i) - L(j)$ is a home for partial limb $L(x) - L(y)$.

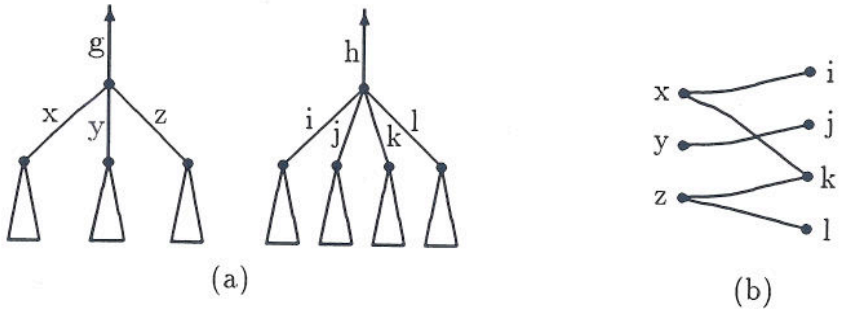


Fig. 4. (a) Limb $L(g)$ of G and limb $L(h)$ of H. Suppose $L(i)$ and $L(k)$ are homes for $L(x)$, $L(j)$ is a home for $L(y)$, and $L(k)$ and $L(l)$ are homes for $L(z)$. (b) The bipartite graph to determine whether $L(h)$ is a home for $L(g)$. $L(h)$ is a home for $L(g)$ since $\{(x,i),(y,j),(z,l)\}$ is a matching that matches all the children of g.

i of h, we know whether or not $L(i)$ is a home for $L(x)$. Construct a bipartite graph B, in which the boys are the child edges of g and the girls are the child edges of h, and there is an edge in B between child x of g and child i of h if and only if $L(i)$ is a home for $L(x)$. Determine if there is a matching in B that matches all the boys in B. $L(h)$ is a home for $L(g)$ if and only if there is such a matching.

Figure 4 gives an example of Procedure P being applied to two limbs.

Theorem 2.1 [15]. *Given two limbs $L(g)$ and $L(h)$, Procedure P correctly determines whether or not $L(h)$ is a home for $L(g)$.*

Thus one can determine whether a limb $L(h)$ is a home for a limb $L(g)$ as follows. If $L(g)$ is a leaf limb, then $L(h)$ is a home for $L(g)$. Else (1) recursively determine which child limbs of h are homes for each of the child limbs of g, and (2) run Procedure P.

We first consider restricted versions of the subtree isomorphism problem. Let *limb-rooted subtree isomorphism* be the subtree isomorphism problem restricted to the case where both G and H are limb-rooted trees, i.e., $G=L(g)$ for some edge g in G and $H=L(h)$ for some edge h in H. Let k be the height of $L(g)$. First, consider a further restriction that g must map to h. One approach to implementing the above recursive technique is to process both G and H bottom-up, level by level, starting with the limbs at level k. In this approach, at each level in turn, the algorithm determines which limbs of $L(h)$ at level i are homes for each of the limbs of $L(g)$ at level i by running Procedure P on each such pair. In this way, at level one, the algorithm determines if $L(h)$ is a home for $L(g)$.

Now consider removing the restriction that the roots must match. One approach is to process $L(g)$ level by level, starting with level k. In this approach, at iteration i, the algorithm determines which limbs of $L(h)$ are homes for the level-i limbs of

- The parent edge has more than one child in $\hat{T}$ and all but one is a leaf edge. In this case, all the children that are leaf edges are deleted from $\hat{T}$ and so the parent becomes a unary edge. The leaf marks on the leaf children are used to compute a unary mark for the parent.
- The parent edge has one child in $\hat{T}$ and it is a leaf edge. Here, the child is deleted from $\hat{T}$ so that the parent becomes a leaf edge. The unary mark on the parent and the leaf mark on the child are used to compute a leaf mark for the parent.

Figure 5 gives an example of a rake operation being applied to a tree.

The second operation in a contract phase is called *compress*. In a compress operation, consecutive edges in unary chains in $\hat{T}$ are paired up, with the pair being replaced by a single edge. In this case, an edge $\hat{e}$ of even parity in its unary chain is paired with its parent edge $\hat{o}$. The unary marks on the two edges are used to compute a new unary mark for the single edge which replaces the original two.

At a general step of the contraction process, each edge of $\hat{T}$ is viewed as corresponding to some edge of the original tree T. Initially, $\hat{T}$ is T, so the correspondence is trivial. At each step, the correspondence can be changed as a result of a compress operation (rake operations do not change the correspondence). Consider an edge $\hat{e}$ of even parity on its unary chain and its parent edge $\hat{o}$, and let $\hat{o}$ correspond to edge o in T. In the tree resulting from applying the compress operation to $\hat{e}$ and $\hat{o}$, these two edges are replaced by a single edge which is considered to correspond to o in T. Edge $\hat{e}$ is considered to have been processed and deleted.

Once T has been contracted to a single edge, a second iterative process can be used to compute final (leaf) marks for *all* the edges of T. An *expansion process* reconstructs T by reversing the contraction process, with each *expand* phase splicing

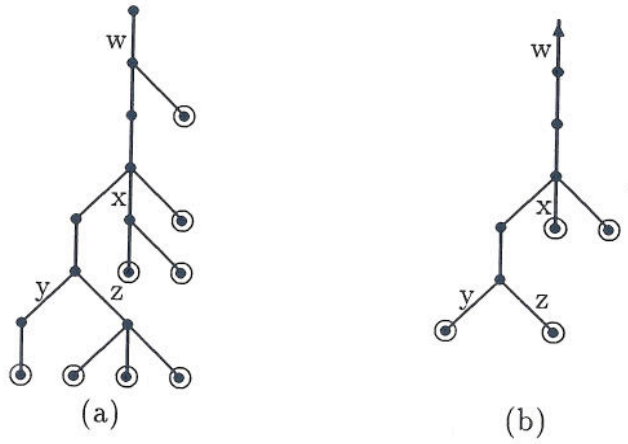


Fig. 5. (a) Limb-rooted tree T, with circles around its leaf nodes. (b) The tree resulting from applying the rake operation to T. Examples of all three types of rake operations are shown: case 1 is applied to edges x and z, case 2 is applied to edge w, and case 3 is applied to edge y.

back into the tree all edges deleted at the corresponding contract phase. Final marks are computed for each edge as it is spliced back into the tree.

For readers familiar with the many variants of the dynamic tree contraction technique (see [11]), we summarize the variant we use:

- We do not assume the tree is binary. In some applications of this technique (e.g. expression evaluation [16]), a nonbinary tree is first converted to an equivalent binary tree, since the technique tends to be easier to apply to binary trees. In our case, we do not know how to convert from a general tree to a binary tree in a way that preserves subtree isomorphism.
- We operate on the edges of the tree, not the nodes.
- We use *rake* with lazy evaluation. In [16], *all* leaves in $\hat{T}$ are deleted at each rake operation. In our variant, we do not delete a leaf unless its parent has at most one nonleaf child. Since this is a necessary condition for the parent to be or become a unary or leaf edge, this particular modification does not affect the number of contract phases required.

Lemma 2.4. *Given a limb-rooted tree T with n edges,* $O(\log n)$ *contract phases are sufficient to reduce T to one edge.*

Proof. Miller and Reif [16] show that their variant of *contract* will reduce a rooted tree to one node in $O(\log n)$ phases, and their proof applies to our variant as well. □

2.3. *Applying the dynamic tree contraction technique*

Our goal is to label each edge of G with a corresponding home edge in H such that these labels define an isomorphic mapping from G to a subtree of H. We use the dynamic tree contraction technique and the ideas on processing leaf and unary edges discussed in Section 2.1. We will maintain a tree $\hat{G}$, derived from G, consisting of edges corresponding to limbs in G that have not yet been processed. $\hat{G}$ starts as the limb-rooted G, but is contracted during the course of the algorithm by a series of contract phases.

We will maintain the following invariants I. At the start of each contract phase, let $\hat{g}_i$ be the edge currently in $\hat{G}$ which corresponds to the limb $L(g_i)$ in G.

(I1) Associated with each leaf edge $\hat{g}_i$ in $\hat{G}$ is a *leaf mark*, which is a set consisting of all possible homes in H for $L(g_i)$.

(I2) Associated with each unary edge $\hat{g}_j$ in $\hat{G}$ is a *unary mark*, which is the set $C(g_j, g_k)$ defined earlier, where $\hat{g}_k$ is the child of $\hat{g}_j$ in $\hat{G}$. Recall that $C(g_j, g_k) = \{(h_w, h_x) \mid L(h_w) - L(h_x)$ is a partial limb in H and $L(h_w) - L(h_x)$ is a home for $L(g_j) - L(g_k)\}$.

We now present a case-by-case description of Procedure *contract*, a contract

phase suitable for subtree isomorphism. In particular, we will show how to implement the compress operation and the three types of rake operations described in Section 2.2 so as to preserve the invariants I. Pseudo-code for our algorithm will be described in Section 3.1.

For each nonleaf edge $\hat{g}_i$ in $\hat{G}$, at most one of the following operations is applied to $\hat{g}_i$ in a contract phase:

(R1) An R1 operation is applied to $\hat{g}_i$ if and only if $\hat{g}_i$ has more than one child in $\hat{G}$ and each child is a leaf edge. The children are deleted from $\hat{G}$ and so $\hat{g}_i$ becomes a leaf edge. For each limb $L(h)$ in H, run Procedure P (of Section 2.1) on $L(g_i)$ and $L(h)$. In setting up the bipartite graph, use the leaf marks on the children of $\hat{g}_i$ to determine which child limbs of h are homes for each of the child limbs of g_i. Place $L(h)$ in the leaf mark for $L(g_i)$ if and only if Procedure P determines that $L(h)$ is a home for $L(g_i)$.

(R2) This operation is applied to $\hat{g}_i$ if and only if $\hat{g}_i$ has more than one child in $\hat{G}$ and all but one child $\hat{g}_j$ is a leaf edge. All the children of $\hat{g}_i$ other than $\hat{g}_j$ are deleted from $\hat{G}$ and so $\hat{g}_i$ becomes a unary edge with child $\hat{g}_j$. Compute a unary mark for $\hat{g}_i$ as follows. For each limb $L(h_w)$ in H, and each of its child limbs $L(h_x)$, run Procedure P' on the guest partial limb $L(g_i)-L(g_j)$ and the host partial limb $L(h_w)-L(h_x)$. In setting up the bipartite graph, use the leaf marks on the children of $\hat{g}_i$ to determine which child limbs of h are homes for each of the child limbs of g_i. Place the ordered pair (h_w, h_x) in $C(g_i, g_j)$ if and only if Procedure P' determines that $L(h_w)-L(h_x)$ is a home for $L(g_i)-L(g_j)$.

(R3) This operation is applied to $\hat{g}_i$ if and only if $\hat{g}_i$ has one child $\hat{g}_j$ in $\hat{G}$ and $\hat{g}_j$ is a leaf edge. The child $\hat{g}_j$ is deleted from $\hat{G}$ and so the parent becomes a leaf edge. Place h_w in the leaf mark for $\hat{g}_i$ if and only if there exists a limb $L(h_x)$ in H such that $(h_w, h_x) \in C(g_i, g_j)$ and h_x is in the leaf mark for $\hat{g}_j$.

(C) The C operation is applied to $\hat{g}_i$ if and only if it is in a unary chain, it is of odd parity on this chain, and is not the last edge on the chain. Let $\hat{g}_j$ be the child of $\hat{g}_i$, and let $\hat{g}_l$ be the child of $\hat{g}_j$ ($\hat{g}_l$ is not a leaf edge). Compose $C(g_i, g_j)$ and $C(g_j, g_l)$ to get a single set $C(g_i, g_l)$ as follows: place the ordered pair (h_w, h_z) in $C(g_i, g_l)$ if and only if there exists a limb $L(h_x)$ such that $(h_w, h_x) \in C(g_i, g_j)$ and $(h_x, h_z) \in C(g_j, g_l)$. Replace the two edges $\hat{g}_i$ and $\hat{g}_j$ by a single edge $\hat{g}_i$ in $\hat{G}$.

Note that only the R1 and R2 operations involve matchings.

Lemma 2.5. *A contract phase deletes from $\hat{G}$ each leaf edge whose parent has at most one nonleaf child and each unary edge of even parity on its unary chain in $\hat{G}$.*

Proof. Follows from inspection of the cases above. □

Lemma 2.6. *Let G be a limb-rooted tree. Let $L(g_i)$ be a limb in G that has t children. Let Procedure contract be applied to G until the tree is contracted to one edge. Prior to each contract phase, let $\hat{G}$ be the tree consisting of edges corre-*

sponding to limbs in G that have not yet been processed. Then (a) *if $\hat{g}_j$ is a child of $\hat{g}_i$ in $\hat{G}$, then g_j is a descendant of g_i in G,* (b) *there are no bipartite matching problems solved for $L(g_i)$ if $t \leq 1$,* (c) *there is exactly one phase in which there are bipartite matching problems solved for $L(g_i)$ if $t > 1$, and* (d) *prior to the matching problems during this phase, $\hat{g}_i$ will have t children in $\hat{G}$, with each child $\hat{g}_j$ corresponding to a child $L(g_j)$ of $L(g_i)$ in G.*

Proof. Claim (a) can be proved by induction on the number of times a C operation is applied to $\hat{g}_i$. Initially, if $\hat{g}_j$ is a child of $\hat{g}_i$ in $\hat{G}$, then $L(g_j)$ is a child limb of $L(g_i)$ in G. Assume the claim is true prior to a next C operation, and let $\hat{g}_j$ be a child of $\hat{g}_i$ in $\hat{G}$. The C operation is the only one that changes a child of $\hat{g}_i$ (others can only delete children), and this operation replaces the current child $\hat{g}_j$ of $\hat{g}_i$ with the current child $\hat{g}_l$ of $\hat{g}_j$. Thus by the inductive assumption, g_l is a descendant of g_j which is a descendant of g_i in G.

As for claim (b), $\hat{g}_i$ has t children in $\hat{G}$ initially since it is a copy of G. Suppose $t \leq 1$. Then only an R3 or C operation can be applied to $\hat{g}_i$, so $\hat{g}_i$ will continue to have at most one child in $\hat{G}$. As neither R1 nor R2 operations are applied to $\hat{g}_i$, there are no bipartite matching problems solved for $L(g_i)$. Claim (b) follows.

If $t > 1$, then the number of children of $\hat{g}_i$ will remain t until the first application of an R1 or R2 operation to $\hat{g}_i$. In both these cases, $\hat{g}_i$ is left with at most one child, and will continue to have at most one child for as long as it remains in $\hat{G}$. Thus this one application of an R1 or R2 operation is the only phase in which there are bipartite matching problems solved for $L(g_i)$.

Claim (d) of this lemma holds since prior to this one application of an R1 or R2 operation, no child of $\hat{g}_i$ can be deleted. □

Lemma 2.7. *A contract phase as defined above* (*i.e.*, R1, R2, R3, *and* C) *preserves the invariants I above.*

Proof. Assume the invariants I are true immediately before the contract phase, and consider an edge $\hat{g}_i$ in $\hat{G}$ of each type. If $\hat{g}_i$ is a leaf edge after contract is applied to $\hat{G}$, then by Lemma 2.5, immediately prior to this contract phase all the children of $\hat{g}_i$ were leaf edges. There are three cases:

- Edge $\hat{g}_i$ was a leaf edge immediately prior to this contract phase. Hence its leaf mark is still valid.
- Edge $\hat{g}_i$ was a unary edge with (leaf) child $\hat{g}_j$. We claim that an R3 operation yields a valid leaf mark for $\hat{g}_i$. If there are limbs $L(h_x)$ and $L(h_w)$ such that $(h_w, h_x) \in C(g_i, g_j)$ ($L(h_x)$ will be a subtree of $L(h_w)$), and h_x is in the leaf mark for $\hat{g}_j$, then by invariants I2 and I1, $L(h_w) - L(h_x)$ is a home for $L(g_i) - L(g_j)$ and $L(h_x)$ is a home for $L(g_j)$. Thus by Lemma 2.3, $L(h_w)$ is a home for $L(g_i)$. Conversely, if $L(h_w)$ is a home for $L(g_i)$, then let g_j be mapped to h_x in an imbedding of $L(g_i)$ in $L(h_w)$. Then partial limb $L(h_w) - L(h_x)$ is a home for $L(g_i) - L(g_j)$,

and thus by invariants I2 and I1, $(h_w, h_x) \in C(g_i, g_j)$ and h_x is in the leaf mark for $\hat{g}_j$.

- Edge $\hat{g}_i$ had more than one child. By Lemma 2.6, the children of $\hat{g}_i$ correspond to the child limbs of $L(g_i)$. By Theorem 2.1, an R1 operation yields a valid leaf mark for $\hat{g}_i$.

Thus invariant I1 holds after the contract phase.

If $\hat{g}_i$ is a unary edge with child $\hat{g}_l$ after contract is applied to $\hat{G}$, then there are two cases to consider:

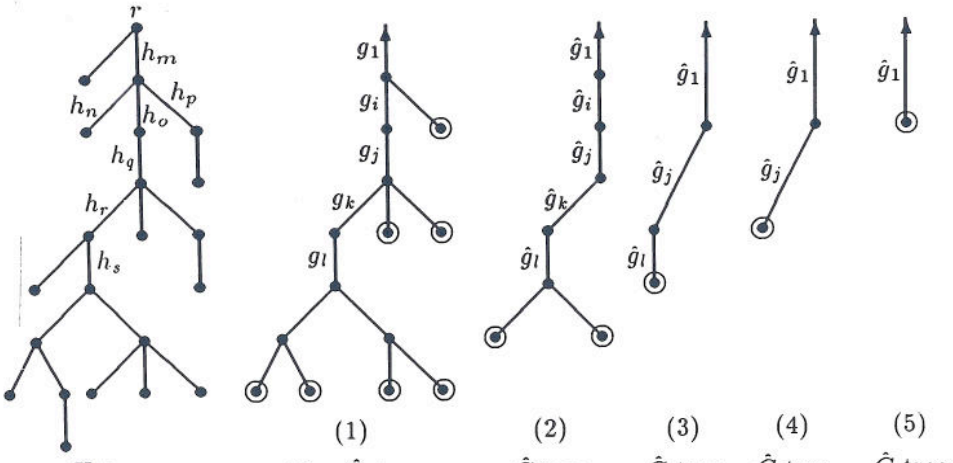
- Edge $\hat{g}_i$ was a unary edge immediately prior to this contract phase. Then by Lemma 2.5, $\hat{g}_i$ was of odd parity in its unary chain since even parity edges are deleted. If $\hat{g}_i$ was the last edge in its unary chain, then its child was $\hat{g}_l$ immediately prior to this contract phase, and so its unary mark is still valid. Else some edge $\hat{g}_j$ was the child edge of $\hat{g}_i$ and the parent edge of $\hat{g}_l$. Thus by Lemma 2.6, g_l is a descendant of g_j which is a descendant of g_i in G. Thus partial limb $L(g_i) - L(g_l)$ is the union of $L(g_i) - L(g_j)$ and $L(g_j) - L(g_l)$. Suppose partial limb $L(h_w) - L(h_z)$ in H is a home for $L(g_i) - L(g_l)$ with a corresponding mapping ϕ from $L(g_i) - L(g_l)$ into $L(h_w) - L(h_z)$. Then ϕ maps g_j to some edge h_x in $L(h_w)$, and using ϕ, we get that $L(h_w) - L(h_x)$ is a home for $L(g_i) - L(g_j)$ and $L(h_x) - L(h_z)$ is a home for $L(g_j) - L(g_l)$. Thus, by invariant I2, there exists a limb $L(h_x)$ such that $(h_w, h_x) \in C(g_i, g_j)$ and $(h_x, h_z) \in C(g_j, g_l)$. Conversely, if there is a limb $L(h_x)$ such that $(h_w, h_x) \in C(g_i, g_j)$ and $(h_x, h_z) \in C(g_j, g_l)$, then by invariant I2 and Lemma 2.3, $L(h_w) - L(h_z)$ is a home for $L(g_i) - L(g_l)$. Thus a C operation yields a valid unary mark for $\hat{g}_i$.
- Edge $\hat{g}_i$ had more than one child, which were all leaf edges except for $\hat{g}_l$. By Lemma 2.6, the children of $\hat{g}_i$ correspond to the child limbs of $L(g_i)$, so the possible homes for $L(g_i) - L(g_l)$ are restricted to partial limbs $L(h_w) - L(h_x)$ such that $L(h_w)$ is in H and h_x is a child edge of h_w in $L(h_w)$. By Corollary 2.2, we can determine whether $L(h_w) - L(h_x)$ is a home for $L(g_i) - L(g_j)$ by applying Procedure P'. Thus an R2 operation yields a valid unary mark for $\hat{g}_i$.

Thus invariant I2 holds after the contract phase, and the lemma follows. □

3. The subtree isomorphism algorithm

In this paper, we presented the limb-rooted version of our algorithm first since it seems easier to picture what is happening as the algorithm progresses. This algorithm can be extended trivially to solve the (unrooted) subtree isomorphism problem. For an unrooted tree, each undirected edge contributes two limbs, one for each way of directing the edge. Each leaf edge in an unrooted tree T defines two limbs: a *leaf limb* consisting of a single edge, and a *root limb* consisting of all of T. In the unrooted case, without loss of generality, we first root G at a root limb. The host tree H is unrooted, but this poses no problem to the parallel algorithm

described in the previous section: the only difference between the rooted and unrooted cases is that in the unrooted case, there are twice as many H limbs to consider when considering all H limbs. In fact, the definition of contract is unchanged, so the lemmas of the previous section hold as stated for the unrooted case. Figure 6 gives an example of contract being applied in the unrooted case.



Phase	limb	status	mark
(1)	g_i	unary with child g_j	$\{(h_o, h_q), \ldots\}$
	g_k	unary with child g_l	$\{(h_r, h_s), \ldots\}$
(2)	$\hat{g}_1$	unary with child $\hat{g}_i$	$\{(h'_p, h_o), \ldots\}$
	$\hat{g}_i$	unary with child $\hat{g}_j$	$\{(h_o, h_q), \ldots\}$
	$\hat{g}_j$	unary with child $\hat{g}_k$	$\{(h_q, h_r), \ldots\}$
	$\hat{g}_k$	unary with child $\hat{g}_l$	$\{(h_r, h_s), \ldots\}$
(3)	$\hat{g}_1$	unary with child $\hat{g}_j$	$\{(h'_p, h_q), \ldots\}$
	$\hat{g}_j$	unary with child $\hat{g}_l$	$\{(h_q, h_s), \ldots\}$
	$\hat{g}_l$	leaf	$\{h_s, \ldots\}$
(4)	$\hat{g}_1$	unary with child $\hat{g}_j$	$\{(h'_p, h_q), \ldots\}$
	$\hat{g}_j$	leaf	$\{h_q, \ldots\}$
(5)	$\hat{g}_1$	leaf	$\{h'_p, \ldots\}$

Fig. 6. Given two unrooted trees G and H, G is rooted at edge g_1 and then contracted using Procedure contract. By convention, we have labeled the edges of H with a single label, as if H were rooted at node r. If $L(h_m)$ is the limb when an edge is directed towards r, then let $L(h'_m)$ denote the limb when the same edge is directed away from r. A circle around a leaf node indicates that a leaf mark has been computed for the edge directed out of the node. For each phase, for a few of the $\hat{G}$ limbs, we show its status and one member of its mark. After phase 5, we conclude that $L(h'_p)$ is a home for the rooted G. In addition, contract would determine that $L(h_m)$ and $L(h'_n)$ are also homes for the rooted G.

Algorithm A: Given two trees, a guest tree G and a host tree H, this algorithm determines if there is a subtree of H isomorphic to G.

1. Let G be rooted so that $G = L(g_1)$ for some edge g_1 in G. Let $\hat{G} = G$.
2. Initialize the **Imbed** and **Unary** matrices to all zeroes. Then for each leaf edge g_k, set **Imbed**$[g_k, h_y]$ to 1 for all directed edges h_y in H. For each unary edge g_i do: for all directed edges h_w, in H, set **Unary**$[g_i, h_w, h_x]$ to 1 for each child h_x of h_w.

3. WHILE there exists > 1 edges in $\hat{G}$ DO {
4. IN PARALLEL for each nonleaf edge $\hat{g}_i$ in $\hat{G}$ DO {
5. IF $\hat{g}_i$ has > 1 child edges in $\hat{G}$ {
6. IF all the children of $\hat{g}_i$ are leaf edges {
7. **(R1)** mark all these child edges for deletion
8. **Determine_Leaf_Mark**$(\hat{g}_i)$
}
9. ELSE IF $\hat{g}_i$ has exactly one nonleaf child $\hat{g}_j$ {
10. **(R2)** mark all its children for deletion except $\hat{g}_j$
11. **Determine_Unary_Mark**$(\hat{g}_i, \hat{g}_j)$
}
}
ELSE { /* let $\hat{g}_j$ be the unique child of $\hat{g}_i$ */
12. IF $\hat{g}_j$ is a leaf edge in $\hat{G}$ {
13. **(R3)** mark $\hat{g}_j$ for deletion
14. determine the leaf mark for $\hat{g}_i$ (i.e. the set of homes for $L(g_i)$) from the unary mark on $\hat{g}_i$ and the leaf mark on $\hat{g}_j$, i.e. if **Unary**$[g_i, h_w, h_x] = 1$ and **Imbed**$[g_j, h_x] = 1$, then set **Imbed**$[g_i, h_w]$ to 1
}
15. ELSE IF $\hat{g}_i$ is of odd parity on its unary chain {
/* let $\hat{g}_l$ be the unique child of $\hat{g}_j$ */
16. **(C)** mark $\hat{g}_j$ for deletion /* $\hat{g}_l$ will be the child of $\hat{g}_i$ */
17. compose the unary marks on $\hat{g}_i$ and $\hat{g}_j$ to get a new unary mark for $\hat{g}_i$, i.e. if **Unary**$[g_i, h_w, h_x] = 1$ and **Unary**$[g_j, h_x, h_z] = 1$, then set (new) **Unary**$[g_i, h_w, h_z]$ to 1
}
}
}
18. IN PARALLEL delete all edges $\hat{g}_j$ that are marked for deletion
}

19. There is a subtree of H isomorphic to G if and only if there exists h_w such that **Imbed**$[g_1, h_w] = 1$

PROCEDURE Determine_Leaf_Mark($\hat{g}_i$):

20. IN PARALLEL for each directed edge h_w of H DO:
21. IF h_w has at least as many children as $\hat{g}_i$ has in $\hat{G}$ {
22. Construct a bipartite graph B in which the boys are the children of $\hat{g}_i$ and the girls are the children of h_w. There is an edge in B between boy $\hat{g}_k$ and girl h_y if and only if $L(h_y)$ is a home for $L(g_k)$, i.e. **Imbed**$[g_k, h_y] = 1$
23. Attempt to find a matching in B that matches all the boys. If one exists, set **Imbed**$[g_i, h_w]$ to 1
}

PROCEDURE Determine_Unary_Mark($\hat{g}_i$, $\hat{g}_j$):

24. IN PARALLEL for each directed edge h_w of H DO:
25. IF h_w has at least as many children as $\hat{g}_i$ has in $\hat{G}$
26. IN PARALLEL for all children h_x of h_w DO {
27. Construct a bipartite graph B' as in A22 above, except exclude children $\hat{g}_j$ and h_x from the graph
28. Attempt to find a matching in B' that matches all the boys. If one exists, set **Unary**$[g_i, h_w, h_x]$ to 1
}

Fig. 7. The subtree isomorphism algorithm. Given a guest tree G and a host tree H, G is first rooted at a root limb. The algorithm operates on H and $\hat{G}$, where $\hat{G}$ starts as the limb-rooted G, but is contracted using rake and compress operations in each iteration of the WHILE loop (A3–A18). Each edge in the unrooted H corresponds to two directed edges; h_w, h_x, h_y, and h_z above denote directed edges in H. At the start of an iteration of the WHILE loop, $\hat{g}_i$, $\hat{g}_j$, $\hat{g}_k$, and $\hat{g}_l$ denote edges currently in $\hat{G}$, and $L(g_i)$, $L(g_j)$, $L(g_k)$, and $L(g_l)$ denote their corresponding limbs in the limb-rooted G. Each leaf edge $\hat{g}_i$ in $\hat{G}$ has a leaf mark, represented as the row **Imbed**$[g_i, -]$. Each unary edge $\hat{g}_j$ in $\hat{G}$ has a unary mark, represented as the matrix **Unary**$[g_j, -, -]$.

3.1. Pseudo-code for the algorithm

To summarize our subtree isomorphism algorithm, we present pseudo-code for the algorithm. Let n_G be the number of nodes in G and $n = n_H$ be the number of nodes in H, where $n_G \leq n_H$. Let G be rooted so that $G = L(g_1)$ for some g_1 in G. Then there are $m_G = n_G - 1$ limbs in the limb-rooted G and $m_H = 2(n_H - 1)$ limbs in H. Our algorithm uses the following two data structures. Let **Imbed**$[-,-]$ be an $m_G \times m_H$ Boolean matrix, used to hold leaf marks. Let **Unary**$[-,-,-]$ be an $m_G \times m_H \times m_H$ Boolean matrix, used to hold unary marks.

Algorithm A gives a pseudo-code description of our subtree isomorphism algorithm (see Fig. 7).

Theorem 3.1. *Given two trees G and H, Algorithm A correctly 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. Initially, each leaf edge $\hat{g}_i$ in $\hat{G}$ corresponds to a leaf limb $L(g_i)$ in G, and thus *any* H limb is a home for $L(g_i)$. Likewise, each unary edge $\hat{g}_i$ with child $\hat{g}_j$ in $\hat{G}$ corresponds to a unary limb $L(g_i)$ in G. Partial limb $L(g_i)-L(g_j)$ is a single edge g_i, so *any* partial limb $L(h_x)-L(h_y)$ where $L(h_y)$ is a child limb of $L(h_x)$ is a home for $L(g_i)-L(g_j)$. Thus the invariants I hold after step A2. By Lemma 2.7 each iteration of the WHILE loop (steps A3–A18) preserves the invariants I. By Lemmas 2.5 and 2.4, the WHILE loop will succeed in reducing $\hat{G}$ to one edge $\hat{g}_1$. By invariant I1, the leaf mark on $\hat{g}_1$ is the set of all possible homes for $L(g_1)$. Therefore, there exists a subtree of H isomorphic to G if and only if there exists a limb $L(h_w)$ such that **Imbed**$[g_1, h_w]=1$. □

3.2. *Implementation details and analysis*

In this section we describe how to implement Algorithm A on a CREW PRAM and how to extend the algorithm to exhibit an isomorphism between G and a subtree of H. For each step which finds a matching in a bipartite graph (i.e., steps A23 and A28), we will use a randomized algorithm due to Mulmuley, Vazirani and Vazirani [17]. We present a detailed analysis of the running time, processor count, and error probability for our (randomized) algorithm. We begin with a discussion of the matching algorithm used, then present a step-by-step analysis of Algorithm A, and finally describe a procedure for constructing an isomorphic mapping of G to a subtree of H.

Matching algorithm. Recall that $M(n)$ is the number of bit operations used by a CREW PRAM to multiply two $n\times n$ Boolean matrices in $O(\log n)$ time ($M(n)\leq n^{2+\varepsilon}$, where ε is less than 0.4 [3, 4]). During the course of the algorithm, for each of a series of bipartite graphs, we find, if possible, a matching that matches all the boys in the graph. In each such matching problem, we first add extra boys to the bipartite graph, with edges to all the girls, in order to make the number of boys equal the number of girls. Then we can apply the Mulmuley, Vazirani, Vazirani randomized algorithm [17] for constructing a *perfect* matching in a bipartite graph. Let B be a bipartite graph with n boys, n girls, and m edges. The algorithm produces a set of edges, which can be checked to see if they form a perfect matching in B. If B does not have a perfect matching, then the algorithm correctly detects this fact. If B has at least one perfect matching, the algorithm finds a perfect matching in B with probability $\geq\frac{1}{2}$. The resource requirements of the algorithm are bounded by the time and processors needed to compute the determinant and adjoint of an $n\times n$ matrix whose entries are $(2m)$-bit (random) integers. This can be done using Pan's algorithm [6, 18] which takes $O(\log^2 n)$ time and $nlM(n)\log\log n/\log n$

processors to invert (with high probability) an $n \times n$ matrix whose entries are l-bit integers. Thus the Mulmuley, Vazirani, Vazirani algorithm takes $O(\log^2 n)$ time and $nmM(n)\log\log n/\log n$ processors, i.e., $o(n^{5.4})$ processors, since $l = 2m \leq 2n^2$.

In order to ensure that an algorithm that solves many bipartite matching problems succeeds with high probability, we increase the success probability of the bipartite matching algorithm by running multiple trials in parallel. In particular, we run $\alpha \log n$ trials of the bipartite matching algorithm in parallel, where α is a constant which depends on the number of bipartite matching problems to be solved and the desired success probability of our algorithm. If a bipartite graph B does not have a perfect matching, then none of the trials will find one. If B has at least one perfect matching, then, with probability $\geq 1 - 1/2^{\alpha \log n} = 1 - n^{-\alpha}$, at least one trial will find a perfect matching. In this case, select any one such perfect matching. Let Algorithm MVV be this modified version of the Mulmuley, Vazirani, Vazirani algorithm. The MVV algorithm runs in $O(\log^2 n)$ time using $n^3 M(n)\log\log n$ processors.

Step-by-step implementation and analysis. In implementing Algorithm A on a CREW PRAM, it is helpful to preprocess H and the limb-rooted G after step A1. For H, use an $m_H \times m_H$ Boolean matrix M initialized to all zeroes. For each pair of limbs h_i and h_j in H, set $M[i,j]$ to one if and only if $L(h_j)$ is a child limb of $L(h_i)$. Using a parallel prefix algorithm [14], compute the index of each child among its siblings. This numbering can be used for allocating edges to processors throughout the algorithm. We preprocess G in the same way.

Here is a step-by-step analysis of Algorithm A. It is convenient to describe the implementation of some steps using PRAM instructions in which multiple processors write to the same location in the same time step (concurrent write). We will later describe how to implement the algorithm on a CREW PRAM, i.e., without concurrent write, in the same asymptotic time and processor bounds.

- For step A1, we root the guest tree G. Given an ordered list of the edges of G, we can find a node of degree one using concurrent write in O(1) time with m_G processors. Having selected a root, we root G using the Euler tour technique for trees [24], in $O(\log m_G)$ time and $m_G/\log m_G$ processors. The preprocessing of G and H that follows takes $O(\log m_H)$ time and $m_H^2/\log m_H$ processors. Given this preprocessing, step A2 takes O(1) time and $m_G m_H^2$ processors, using one processor per matrix entry.
- Steps A4–A18 perform one contract phase. The tests in steps A3–A6, A9, A12, and A15 depend on the structure of the current tree. In each case, we wish to determine if an edge has zero, one, or more than one child edges of a particular type. The most time-efficient way to perform these tests is using concurrent write. Each edge $\hat{g}_j$ in $\hat{G}$ with parent edge $\hat{g}_i$ writes j in cell i, then reads cell i to see if it has succeeded in its write attempt. If not, it complains to its parent. This takes O(1) time and m_G processors. For step A15, each unary edge must determine its parity in its

unary chain (if any). This can be done in $O(\log m_G)$ time with m_G processors: the index of each node in a chain is computed by $O(\log m_G)$ applications of pointer jumping.

- Steps A7, A10, A13, and A16, i.e., statements R1, R2, R3, and C, can be done in O(1) time using m_G processors. Likewise, step A18 takes O(1) time and m_G processors. For statements R1 and R2, all the leaf child edges read from their parent, in order to see which parent is ready to have all its children mark themselves for deletion.
- Using concurrent write, step A14 takes O(1) time and m_H^2 processors for each g_i. For step A17, perform a Boolean matrix multiplication for each g_i in $O(\log m_H)$ time and $M(m_H)/\log m_H$ processors. Note that a temporary matrix is helpful here, since **Unary** is updated in place.
- For step A8, i.e., steps A20–A23, for each edge g_i, we find matchings in parallel for at most m_H bipartite graphs, each with at most $n_H - 2$ girls. (Note that for each edge with fewer than two children in G, we will not solve any bipartite matching problems.) In order to apply the MVV algorithm, we first add extra boys to the graph, with edges to all the girls, so as to have the same number of boys as girls. Using the MVV algorithm, step A23 takes $O(\log^2 n_H)$ time and $m_H n_H^3 M(n_H) \log\log n_H$ processors for all matchings for each edge g_i. Steps A21 and A22 use the preprocessing information obtained for H and G, as well as the **Imbed** matrix, to set up the adjacency matrices for the bipartite graphs.
- Similarly, for step A11, i.e., steps A24–A28, for each edge g_i, we find matchings in parallel for at most $m_H(n_H - 2)$ bipartite graphs, each with at most $n_H - 2$ girls. Thus step A28 takes $O(\log^2 n_H)$ time and $m_H n_H^4 M(n_H) \log\log n_H$ processors for all matchings for each edge g_i.
- Step A19 can be done in O(1) time and m_H processors using concurrent write.

By Lemma 2.4, there will be $O(\log m_G)$ iterations of the WHILE loop. Not counting steps A8 and A11, the algorithm runs in $O(\log m_G \log m_H)$ time on a CRCW PRAM with $m_G M(m_H)/\log m_H$ processors. The time for the steps above that have been described using concurrent write is only $O(\log m_G)$. Thus using a standard simulation of a CRCW PRAM by a CREW PRAM (see [11]) on each of the steps involving concurrent write yields an $O(\log m_G \log m_H)$ time algorithm with the same number of processors. By Lemma 2.6, step A8 or step A11 will be executed at most once for each g_i. It follows that Algorithm A runs in $O(\log n_G \log^2 n_H)$ time on a CREW PRAM with $n_G n_H^5 M(n_H) \log\log n_H$ processors, i.e. $o(n_G n_H^{7.4})$ processors. In Section 4, we show how the processor count can be significantly reduced. Given two trees G and H such that G is not isomorphic to a subtree of H, Algorithm A will correctly determine this fact. Given two trees G and H such that G is isomorphic to a subtree of H, Algorithm A will correctly determine this fact with probability $\geq 1 - 1/n$. The algorithm solves fewer than n^3 bipartite matching problems, and so this success probability can be achieved using the MVV algorithm with $\alpha = 4$ (i.e., perform each matching computation $4 \log n$ times in parallel).

Algorithm A (enhancements): These steps are added to algorithm A.

10a. and for all its children g_k (except g_j), set g_k.remaining_sibling to g_j

14a. and set **Save_Imbed**$[g_j, h_w]$ to h_x

16a. and set g_j.child_when_deleted to g_l

17a. and set **Save_Unary**$[g_j, h_w, h_z]$ to h_x

18a. and save the "time" and "type" (i.e. R1, R2, R3, or C) of deletion. Set g_j.parent_when_deleted to g_i

23a. and save the matching M as follows: if $\hat{g}_k$ is matched with h_y in M, set **Save_Imbed**$[g_k, h_w]$ to h_y

28a. and save the matching M: if $\hat{g}_k$ is matched with h_y in M, set **Save_Unary**$[g_k, h_w, h_x]$ to h_y

PROCEDURE Expand_Tree: This procedure is run after G has been contracted to one edge $\hat{g}_1$. Let $L(h_1)$ be the home for $L(g_1)$, i.e. let h_1 be g_1.home.

29. Let t step by -1 from the number of **contract** phases down to 1
30. IN PARALLEL for each edge g_j in G DO:
31. IF g_j was deleted at time t {
/* Let h_w be (g_j.parent_when_deleted).home */
32. splice $\hat{g}_j$ back into $\hat{G}$

33. IF g_j was marked for deletion by statement R1 or R3
34. g_j.home $\leftarrow$ **Save_Imbed**$[g_j, h_w]$

35. ELSE IF g_j was marked for deletion by statement R2
36. g_j.home $\leftarrow$ **Save_Unary**$[g_j, h_w, h_x]$,
where h_x is (g_j.remaining_sibling).home

ELSE /* g_j was marked for deletion by statement C */
37. g_j.home $\leftarrow$ **Save_Unary**$[g_j, h_w, h_z]$,
where h_z is (g_j.child_when_deleted).home
}

Fig. 8. Together with Algorithm A, these instructions construct an isomorphic mapping from G to a subtree of H (if one exists).

Constructing an isomorphism. In order to exhibit an explicit isomorphism of G to a subtree of H, we make the following enhancements to Algorithm A. While contracting the tree, count the number of contract phases applied so far, in order to save the "time" each edge was deleted from $\hat{G}$. Depending on the "type" of the deletion (i.e., R1, R2, R3, or C), also save the name of its parent when the edge was deleted, the name of its child when deleted, and/or the name of the nonleaf sibling. Save all perfect matchings constructed, and for each matrix entry which is set to one in a new unary or leaf mark, save the name of a corresponding home for the deleted edge. The precise instructions added to Algorithm A are listed in Fig. 8 (shown properly indented to fit into Algorithm A). **Save_Imbed** is an $m_G \times m_H$ matrix and **Save_Unary** is an $m_G \times m_H \times m_H$ matrix.

Given the above enhancements to Algorithm A, Procedure Expand_Tree shown in Fig. 8 can be used to exhibit the mapping. After G has been contracted, we reconstruct G by an expansion process which reverses the contraction process, with each expand phase splicing back into $\hat{G}$ all edges deleted at the corresponding contract phase. At the conclusion of each expand phase, we will have computed the home for each limb $L(g_i)$ in G corresponding to an edge $\hat{g}_i$ in the current $\hat{G}$. Because they are associated with limbs in G, these home edges typically will be scattered throughout H prior to the final expand phase. During the expansion process, new homes are computed based on both the matchings performed during the contraction process and the homes of existing edges in $\hat{G}$.

Clearly the time and processor count for Procedure Expand_Tree is bounded by the time and processor count for Algorithm A.

4. Processor efficiency

We have recast the subtree isomorphism problem as a problem on limbs, in order to save having to try out all possible roots for the trees. In this section, we describe techniques for further reducing the number of processors used by our algorithm. First, we will show how to use an algorithm for *deciding* whether a perfect matching exists while contracting $\hat{G}$, and an algorithm for *constructing* the matching while expanding $\hat{G}$. This reduces the number of processors used since (1) the fastest known parallel decision algorithm for bipartite matching uses fewer processors than the fastest known parallel search algorithm, and (2) the expansion process needs fewer matching problems solved (in parallel) than the contraction process. (There are, however, certain advantages to constructing the matchings as we contract $\hat{G}$: see Section 6.) Second, we will show how the solution to a single bipartite matching problem can yield the solution to a group of related matching problems.

In analyzing the processor bounds for our algorithm, we will often use the following (common) approach. First, describe the algorithm using a convenient (but perhaps wasteful) number of processors. Then determine the *work* of the algorithm, where the *work* of an algorithm is defined to be the sum over all processors p_i of

the number of PRAM instructions executed by p_i during the course of the algorithm. Finally, apply Brent's scheduling lemma [2] to determine the actual number of processors needed. The scheduling lemma states that an algorithm running in time t with work w can achieve time $O(t)$ using only w/t processors, provided that there is negligible overhead in both determining the amount of work to be done at each step of the original algorithm and scheduling this work among the w/t processors. Typically, this allocation of work to processors is predetermined and thus creates no overheads to the algorithm.

4.1. Constructing the matchings while expanding $\hat{G}$

We can use a *decision* algorithm for the matching problems in steps A23 and A28, if we make the following modifications to Expand_Tree. We will construct any necessary matchings while expanding the tree. As before, we will maintain the invariant that the home is known for every edge currently in $\hat{G}$. At the beginning of each expand phase, if (decision) bipartite matching problems were solved for edge $\hat{g}_i$ at the corresponding contract phase, construct the appropriate matching (described below) and save the results in **Save_Imbed** and **Save_Unary**. There are two cases to consider. (1) If step A8 was performed for $\hat{g}_i$, then the home $L(h_w)$ of $\hat{g}_i$ is known, so it suffices to solve only one (search) bipartite matching problem for $\hat{g}_i$: the matching problem between the child edges of $\hat{g}_i$ and the child edges of h_w. (2) If step A11 was performed for $\hat{g}_i$, then both the home h_w for $\hat{g}_i$ and the home h_x for the remaining child $\hat{g}_j$ of $\hat{g}_i$ are known, so it suffices to solve only one (search) bipartite matching problem for $\hat{g}_i$: the matching problem between the child edges of $\hat{g}_i$ other than $\hat{g}_j$ and the child edges of h_w other than h_x.

Lemma 4.1. *Let $L(h_w)$ be a home for $L(g_1)$. During the expansion process, there is at most one bipartite matching constructed for each edge in $L(h_w)$ with more than one child, and no bipartite matchings constructed for any other edge in H.*

Proof. By Lemma 2.6, for each edge g_i, either step A8 is performed once, step A11 is performed once, or neither are performed. Thus by the remarks above, there will be at most one bipartite matching problem solved for each edge g_i during the expansion process, and hence at most one solved for the home edge for g_i. Furthermore, if an edge in H has fewer than two children, then it is not involved in any bipartite matching problems. □

The running time for expanding the tree as described above is $O(\log n_G \log^2 n_H)$, using the MVV algorithm for constructing perfect matchings. Clearly the work to expand the tree is dominated by the work to construct the matchings. From Section 3.2, we see that the work to construct one perfect matching in a bipartite graph with n boys and n girls using the MVV algorithm is $O(n^3 M(n) \log^2 n \log\log n)$. Let $\{h_1, h_2, \ldots, h_k\}$ be the edges in $L(h_w)$, the home for $L(g_1)$, and let d_j be

the number of child edges of h_j in $L(h_w)$. By Lemma 4.1, the work is at most $\sum_{j=1}^{k} d_j^3 M(d_j)\log^2 d_j \log\log d_j$, i.e., the work is $O(n_H^3 M(n_H)\log^2 n_H \log\log n_H)$. Thus the work to expand the tree is, to within a constant factor, the same as the work done by the MVV algorithm to solve *one* matching problem on a bipartite graph with n_H nodes on each side. The algorithm solves fewer than $\frac{1}{2}n$ bipartite matching problems while expanding the tree, and so we will use the MVV algorithm with $\alpha = 2$ (recall from Section 3.2 that we perform $\alpha \log n$ trials of the matching algorithm in parallel). Thus with probability $\geq 1 - 1/2n$, the algorithm will correctly construct an isomorphic mapping between G and a subtree of H.

4.2. *Contracting $\hat{G}$ using fewer processors*

We will now describe and analyze a method for contracting the tree $\hat{G}$ using a factor of $\Omega(n^3)$ fewer processors than the method described in Section 3. First, we will use the following *decision* algorithm for bipartite matching while contracting $\hat{G}$. Given a bipartite graph B with n boys and n girls, the adjacency matrix for B is an $n \times n$ matrix C such that the element in row i, column j of C is one if there is an edge between boy i and girl j, and zero otherwise. Let C' be the matrix obtained from C by replacing each nonzero entry of C with a unique indeterminate x_{ij}. Then the determinant of C' is nonzero if and only if there is a perfect matching in B [5]. If B has a perfect matching, then the determinant is a degree-n polynomial f on up to n^2 variables, where f is not identically 0. If we plug in for each indeterminate in C' an integer chosen uniformly at random from the range $(0, \ldots, \kappa n^3)$, then the determinant of C' will be nonzero with probability $1 - 1/\kappa$ [22]. For our purposes, it suffices to let κ be polynomial in n. Based on this fact, Borodin, von zur Gathen and Hopcroft [1] developed a randomized algorithm for deciding if a bipartite graph has a perfect matching that runs in $O(\log^2 n)$ time on a CREW PRAM. An improved version of their algorithm computes determinants over Z_p, the integers modulo some suitable prime p of magnitude $O(\kappa n^4)$ [21]. This can be done with $O(\sqrt{n}M(n))$ work on a CREW PRAM, using the Preparata and Sarwate algorithm [20] for computing the adjoint and the determinant of a matrix, since all operations involve $O(\log n)$-bit numbers. (Galil and Pan [6] have an algorithm for inverting matrices over Z_p with slightly less work). Let Algorithm S be this improved method for deciding if a bipartite graph has a perfect matching.

While contracting the tree, we can further save processors by solving groups of related matching problems at once. In particular, we can efficiently test, by solving only one matching problem, whether $\hat{g}_i$ in $\hat{G}$ is imbeddable in each of the limbs associated with a node v in H, i.e., those limbs $L(h_w)$ where h_w is directed *out* of v. (Recall that H is unrooted.) Matula [15] showed how to perform such tests efficiently on a sequential machine. In what follows, we present a parallel implementation which results in additional processor savings for our algorithm. Construct a bipartite graph B in which the boys are the child edges of $\hat{g}_i$ in $\hat{G}$ and the girls are the (directed) edges h'_w in H directed *into* v. Add extra boys in order to

equal the number of girls, as was done for step A23 (see Section 3.2). Run Algorithm S on the graph B to compute the appropriate adjoint matrix D. Each entry of D contains the determinant of some minor of D (a *cofactor*). In particular, the entry in row i, column j contains the determinant of the submatrix of D that results from removing row j and column i from D. Thus by testing whether a cofactor is nonzero, we can determine if a perfect matching exists when $\hat{g}_i$ and any one edge directed into v are left out of the bipartite graph. Let r be an extra boy in B (there is at least one such boy since the number of edges directed into v must be larger than the number of children of $\hat{g}_i$ in order for $L(g_i)$ to be imbeddable in a limb directed out of v). $L(g_i)$ is imbeddable in $L(h_w)$ (a limb directed *out* of v) if and only if the cofactor associated with girl h'_w (an edge directed into v) and boy r is nonzero. From Rabin and Vazirani [21], it follows that this holds even when the adjoint is computed over Z_p.

We now analyze the work for contracting the tree using the above approach. Let $\{v_1, v_2, \ldots, v_{n_H}\}$ be the nodes in H and let d_i be the degree of node v_i. Then for each edge g_i, step A8 contributes at most n_H bipartite matching problems (one per each v_i in H), each with at most d_i girls. The work for solving these matching problems is at most $n_G \sum_{i=1}^{n_H} \sqrt{d_i} M(d_i)$, i.e., the work is $O(n_G \sqrt{n_H} M(n_H))$. The cofactor technique can be applied to step A11 as well, where the parent edges $\hat{g}_i$ and h_w are left out of the graph entirely and the cofactors are used to determine if a perfect matching exists when the remaining child $\hat{g}_j$ and any one edge directed into v are ignored. Then for each edge g_i, step A11 contributes m_H bipartite matching problems, each with fewer than n_H girls. Thus the work for solving all step A11 matching problems is $O(n_G m_H \sqrt{n_H} M(n_H))$. The algorithm solves fewer than $m_G m_H$ bipartite matching problems while contracting the tree, and so we will use Algorithm S with $\kappa = 4n^3$. Thus with probability $\geq 1 - 1/2n$, the algorithm will correctly determine if G is isomorphic to a subtree of H.

4.3. *Analysis of the improved version of our algorithm*

From the previous sections, we see that the work for our algorithm is dominated by the work for expanding the tree, which is within a constant multiple of the work for solving a single bipartite matching problem using Algorithm MVV. Given this analysis of the work, we can apply Brent's scheduling lemma to determine the number of processors needed. Before each phase i, 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 less than a factor of two. Let Algorithm A' be the improved version of Algorithm A which uses the above steps to save processors and to construct an isomorphic mapping from G to a subtree of H. Then the following theorem follows from Theorem 3.1, the correctness and analysis of the matching algorithms, and inspection of the cases involved in expanding the tree.

Theorem 4.2. *Given two trees G and H such that G is not isomorphic to a subtree of H, Algorithm A′ will correctly determine this fact. Given two trees G and H such that G is isomorphic to a subtree of H, Algorithm A′ will correctly construct an explicit isomorphism of G to a subtree of H with probability* $\geq 1 - 1/n$. *Algorithm A′ runs in* $O(\log^3 n)$ *time on a CREW PRAM with* $n^3 M(n) \log\log n / \log n$ *processors, where n is the number of nodes in H.*

5. Reducing matching to subtree isomorphism

In this section we show that bipartite perfect matching is log-space reducible to subtree isomorphism. Let $B=(X, Y, E)$ 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 B. 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 B. V_1 and V_2 have no children. For $1 \leq i \leq n$, X_i is the parent of i children, X_{ij}, 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 B. U_1 and U_2 have no children. For $1 \leq i \leq n$, Y_i is the parent of n children, Y_{ij},

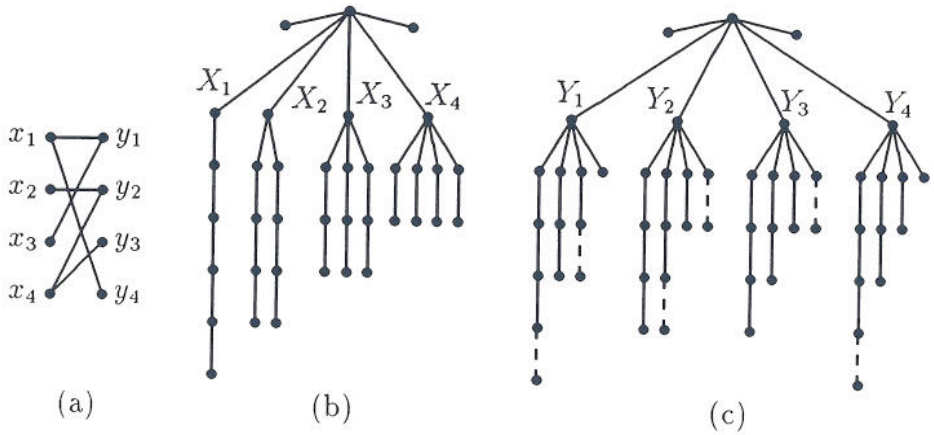


Fig. 9. A log-space reduction from bipartite perfect matching to subtree isomorphism. (a) A bipartite graph B. (b) The tree T_X when $n=4$. (c) The tree T_Y derived from B. The solid lines in T_Y are present for any bipartite graph with four boys and four girls. For each particular edge in B, a dashed line is added as shown. By construction, tree T_X is imbeddable in tree T_Y if and only if B has a perfect matching. In this example, T_X is imbeddable in tree T_Y with X_1 imbedded in Y_4, X_2 in Y_2, X_3 in Y_1, and X_4 in Y_2, which corresponds to a perfect matching in B.

where Y_{ij} is the root of a path of length $n-j+1$ if $\{y_i, x_j\} \in E$ and length $n-j$ otherwise.

An example is given in Fig. 9. Note that this reduction can clearly be performed in logarithmic space.

Lemma 5.1. *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 $Y_{j1}, \ldots, Y_{j,i-1}$ 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_{ji} 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 5.2. *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 5.3. *T_X is imbeddable in T_Y if and only if B has a perfect matching.*

Proof. Let $M = \{\{x_1, y_{\sigma(1)}\}, \ldots, \{x_n, y_{\sigma(n)}\}\}$ be a perfect matching of B. By Lemma 5.1, the subtree rooted at X_i is imbeddable in the subtree rooted at $Y_{\sigma(i)}$ 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 5.2, R_X is mapped to R_Y, and it follows that X_i is mapped to some $Y_{\sigma(i)}$ for each i. By Lemma 5.1, $\{x_i, y_{\sigma(i)}\} \in E$ for all i, and thus the set of edges $\{\{x_1, y_{\sigma(1)}\}, \ldots, \{x_n, y_{\sigma(n)}\}\}$ constitutes a perfect matching of B. □

Corollary 5.4. *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 5.5. *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.*

Theorem 5.6. *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 B.*

Proof. A perfect matching, $M = \{\{x_1, y_{\sigma(1)}\}, \ldots, \{x_n, y_{\sigma(n)}\}\}$, of B induces a unique mapping of X_i to Y_j. The subtree rooted at X_i can be imbedded in exactly $i!$ ways

into the subtree rooted at $Y_{\sigma(i)}$. The vertices V_1, V_2 can be mapped in two ways to U_1, U_2. The theorem follows. □

Corollary 5.7. *The problem of determining the number of imbeddings of a tree in another tree is #P-complete.*

6. Remarks

Applying the standard simulation of uniform PRAM programs by uniform Boolean circuit families [23] to our $O(\log^3 n)$ algorithm yields a uniform family of *unbounded* fan-in circuits (with random inputs) of depth $O(\log^3 n)$ for subtree isomorphism, and hence a uniform family of *bounded* fan-in circuits (with random inputs) of depth $O(\log^4 n)$ for the problem. Since our algorithm uses a polynomial number of processors, the resulting circuit family is of polynomial size, and hence we have placed subtree isomorphism in RNC^4. However, with appropriate implementation of our algorithm, we can place subtree isomorphism in RNC^3. To see this, first observe that without the matchings, our algorithm runs in $O(\log^2 n)$ time, and thus the standard simulation yields an RNC^3 circuit family. Second, the bipartite matching algorithms used are known to be in RNC^2, and we apply them in at most $O(\log n)$ phases. It follows that our algorithm is in (Boolean) RNC^3.

The case where G (or H) has bounded maximum degree d can be done *deterministically* in $O(d \log d \log n)$ time on a CRCW PRAM. Simply solve each bipartite matching problem in our algorithm using d applications of a parallel augmenting path algorithm (an augmenting path can be constructed using a breadth first search calculation on the graph that results from directing all matched edges from boys to girls and all unmatched edges from girls to boys). Each such application requires only $O(\log d)$ time (using concurrent write) since there are at most d boys in the graph and hence any path is at most $2d$ long.

A number of variants on our basic subtree isomorphism algorithm are possible. These differ principally in various implementation details used for solving bipartite matching problems during the course of the algorithm. The full details are given in [9].

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, then the Mulmuley, Vazirani, Vazirani algorithm uses $M(n)$ work. This yields a randomized subtree isomorphism algorithm that uses $nM(n)/\log^2 n$ processors and runs in $O(\log^3 n)$ time.

We can extend our algorithm to a *Las Vegas* algorithm, i.e., an algorithm that runs in expected time t and always produces the correct answer, as follows. While contracting the guest tree, we use a version of the Mulmuley, Vazirani, Vazirani algorithm which constructs a *maximum* matching [17] with high probability. We test

whether the matching constructed is indeed a maximum matching by testing for an augmenting path. We repeat until the randomized algorithm yields a correct maximum matching. This Las Vegas version of our algorithm runs in expected time $O(\log^3 n)$ with $n^4 M(n)\log\log n/\log n$ processors, since we must construct the matchings while contracting the tree.

Finally, we discuss two variations on the subtree isomorphism problem that occur in practice. In the area of pattern recognition, the two trees are often rooted and labeled with attributes at each node. For all nodes v in a rooted tree T, other than the root, associate the label at v with the edge directed out of v in T. The algorithm presented in this paper can be trivially extended to this labeled problem. When computing the *first* (unary or leaf) mark for a limb g in G (i.e., steps A2, A8, A11, or A17 in Algorithm A), consider the additional restriction that a limb $L(h)$ in H can be a home for $L(g)$ only if g and h have compatible labels. Another version of the problem that arises in practice is the case where the trees have a fixed planar orientation, i.e., the trees are rooted and the children at each node v have a fixed ordering $(v_1, v_2, \ldots, v_k)$. We wish to determine whether there is a subtree of H that is isomorphic to G such that the isomorphic mapping preserves the orientation. In particular, if $g \in G$ with ordered children $(g_1, \ldots, g_k)$ is mapped to $h \in H$ with ordered children $(h_1, \ldots, h_l)$, then each child g_i is mapped to $h_{\sigma(i)}$ where $1 \le \sigma(1) < \sigma(2) < \sigma(3) < \cdots < \sigma(k) \le l$. Recently, Gibbons, Miller and Teng [10] developed a deterministic algorithm for this oriented version of the subtree isomorphism problem. Their algorithm runs in $O(\log^2 n)$ time on an EREW PRAM with $n^2/\log^2 n$ processors.

Acknowledgement

A preliminary version of this paper appeared in [8]. The authors would like to thank an anonymous referee whose detailed comments improved the presentation of this paper.

The first author was supported by the International Computer Science Institute (ICSI), Berkeley, CA and by an IBM predoctoral fellowship, while 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, CA. This work was completed while the fourth author was a graduate student at Berkeley and 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.

References

[1] A. Borodin, J. von zur Gathen and J.E. Hopcroft, Fast parallel matrix and GCD computations, in: Proceedings of the Symposium on Foundations of Computer Science (FOCS) (1982) 65–71.

[2] R.P. Brent, The parallel evaluation of general arithmetic expressions, J. ACM 21 (1974) 201–208.
[3] D. Coppersmith, personal communication, 1988.
[4] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, in: Proceedings of the Symposium on Theory of Computing (STOC) (1987) 1–6.
[5] J. Edmonds, Systems of distinct representatives and linear algebra, J. Res. Nat. Bur. Standards 71 (1967) 241–245.
[6] Z. Galil and V. Pan, Improved processor bounds for algebraic and combinatorial problems in RNC, in: Proceedings of the Symposium on Foundations of Computer Science (FOCS) (IEEE Press, New York, 1985) 490–495.
[7] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness (Freeman, New York, 1979).
[8] P.B. Gibbons, R.M. Karp, G.L. Miller and D. Soroker, Subtree isomorphism is in random NC, in: J.H. Reif, ed., Proceedings of the 3rd Aegean Workshop on Computing, Lecture Notes in Computer Science 319 (Springer, Berlin, 1988) 43–52.
[9] P.B. Gibbons, R.M. Karp, G.L. Miller and D. Soroker, Subtree isomorphism is in random NC, Tech. Rept. TR-89-014, International Computer Science Institute, Berkeley, CA (1989).
[10] P.B. Gibbons, G.L. Miller and S.-H. Teng, personal communication, 1988.
[11] R.M. Karp and V.L. Ramachandran, A survey of parallel algorithms for shared-memory machines, Tech. Rept. UCB-CSD-88-408, Computer Science Division, University of California, Berkeley, CA (1988); also in: Handbook of Theoretical Computer Science (North-Holland, Amsterdam, to appear).
[12] R.M. Karp, E. Upfal and A. Wigderson, Constructing a perfect matching is in random NC, Combinatorica 6 (1) (1986) 35–48.
[13] M. Karpinski and A. Lingas, Subtree isomorphism and bipartite perfect matching are mutually NC reducible, Submitted.
[14] R.E. Ladner and M.J. Fischer, Parallel prefix computation, J. ACM 27 (4) (1980) 831–838.
[15] D.W. Matula, Subtree isomorphism in $O(n^{5/2})$, Annals of Discrete Mathematics 2 (North-Holland, Amsterdam, 1978) 91–106.
[16] G.L. Miller and J.H. Reif, Parallel tree contraction and its applications, in: Proceedings of the Symposium on Foundations of Computer Science (FOCS) (1985) 478–489.
[17] K. Mulmuley, U.V. Vazirani and V.V. Vazirani, Matching is as easy as matrix inversion, Combinatorica 7 (1) (1987) 105–113.
[18] V. Pan, Fast and efficient parallel algorithms for the exact inversion of integer matrices, in: S. Maheshwari, ed., Proceedings of the 5th Foundations of Software Technology and Theoretical Computer Science Conference, Lecture Notes in Computer Science 206 (Springer, Berlin, 1985) 504–521.
[19] N. Pippenger, On simultaneous resource bounds, in: Proceedings of the Symposium on Foundations of Computer Science (FOCS) (1979) 307–311.
[20] F.P. Preparata and D.V. Sarwate, An improved parallel processor bound in fast matrix inversion, Inform. Process. Lett. 7 (3) (1978) 148–150.
[21] M.O. Rabin and V.V. Vazirani, Maximum matchings in general graphs through randomization, Tech. Rept. TR-15-84, Aiken Computation Laboratory, Harvard University, Cambridge, MA (1984).
[22] J.T. Schwartz, Fast probabilistic algorithms for verification of polynomial identities, J. ACM 27 (1980) 701–717.
[23] L.J. Stockmeyer and U. Vishkin, Simulation of parallel random access machines by circuits, SIAM J. Comput. 13 (1984) 409–422.
[24] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, SIAM J. Comput. 14 (1985) 862–874.