Optimal Speedups for Parallel Pattern Matching in Trees[†]

R. Ramesh and I.V. Ramakrishnan

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract

Tree pattern matching is a fundamental operation that is used in a number of programming tasks such as code optimization in compilers, symbolic computation, automatic theorem proving and term rewriting. An important special case of this operation is linear tree pattern matching in which an instance of any variable in the pattern occurs at most once. If n and m are the number of nodes in the subject and pattern tree respectively and if no restriction is placed on the structure of the trees, then the fastest known sequential algorithm for linear tree pattern matching requires $O(nm)$ time in the worst case.

In this paper we present a parallel algorithm for linear tree pattern matching on a PRAM (parallel random access machine) model. Our algorithm exhibits optimal speedup, in the sense that its processor-time product matches the worst-case time complexity of the fastest sequential algorithm.

1 Introduction

Tree replacement systems are used in a number of significant programming tasks such as design of interpreters for nonprocedural programming languages[9], automatic implementations of abstract data types [7], code optimization in compilers[1,11], symbolic computations[3], context searching in structure editors, automatic theorem proving and term rewriting. A key operation in tree replacement systems involves simplifying trees by repeatedly replacing subtrees according to a set of replacement rules. A fundamental and compute-intensive step in this operation involves repeated searching for subtrees which may be replaced. This is essentially a *tree-pattern-matching* problem.

Informally, given a subject tree s and a pattern tree(s) p, the tree-pattern-matching problem is concerned with identifying the nodes in s such that the subtrees rooted at these nodes are identical to p following replacement of nodes labeled with variables in p by appropriate subtrees of s. For example, consider the tree s and pattern p shown in Figure 1. p has two nodes labeled with variables X and Y. Observe that p matches s at the root since replacing X by the subtree rooted at node 2 and Y by c makes p and s identical. Similarly, it can be easily verified that p also matches s at node 2.

Due to the importance of this problem, several researchers have examined the design of sequential algorithms for it. Karp, Miller and Rosenberg [10] were the first to give an algorithm for this problem. However, they required the patterns to be full binary trees. This problem was later examined by Hoffmann and O'Donnell [8]. They presented a bottom-up and a top-down algorithm for linear tree pattern matching. In a linear pattern an instance of any variable occurs at most once. If n and m are the total number of nodes in the subject and pattern tree respectively, then their bottom-up algorithm has a worst case complexity of $O(2^m + n)$ whereas the top-down algorithm requires $O(mn)$ time in the worst case. Recently, an efficient implementation of their bottom-up algorithm for non-linear patterns was described by Purdom and Brown [13]

Since tree pattern matching is a commonly repeated operation in tree replacement systems, it is important to speed up this operation. One way to speed it up is to use parallel processors. The standard model of synchronous parallel computation is a PRAM (parallel random access machine) [5]. This model consists of a collection of processors that share a global memory and execute a single program in lock step. There are variants of the PRAM model which handle conflicting reads and writes to the

[†]Research supported in part by NSF under grant number ECS-84-04399 and in part by ONR under contract N00014-84-K-0530

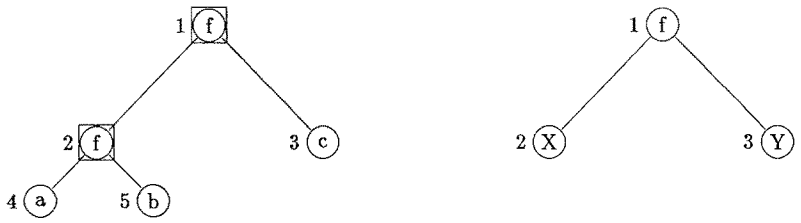

Figure 1: Example 1

same global memory location differently. The strongest variant is the CRCW PRAM (Concurrent Read Concurrent Write) model that allows simultaneous reading and writing to the same location by more than one processor. The CREW PRAM (Concurrent Read Exclusive Write) allows simultaneous reading by more than one processor, but disallows simultaneous writes. The weakest variant is the EREW PRAM (Exclusive Read Exclusive Write) in which simultaneous reading and simultaneous writing are both prohibited.

The main goal in the design of parallel algorithms on a PRAM is to construct an algorithm for a problem such that its processor-time product is asymptotically equal to the worst-case time complexity of the fastest known sequential algorithm for the same problem. Such an algorithm is said to exhibit *optimal* speed up [18].

In this paper we present a deterministic parallel algorithm for linear tree pattern matching. To the best of our knowledge this is the first known algorithm for this problem on the PRAM model. Our parallel algorithm requires an entirely new approach than those used in sequential algorithms. For instance, the two methods employed in [8] are inherently sequential. This is because the top-down approach requires a preorder traversal of the subject tree while in the bottom-up approach a match for an internal node cannot be computed before computing the match for all of its descendants.

The main results in this paper are as follows. Let n and m be the size of the subject and pattern tree respectively. Let k be the number of variables associated with the nodes of the pattern tree.

1. We describe a parallel algorithm on EREW PRAM that uses $O(n^{2-\epsilon})$ processors and requires $O(n^{\epsilon} log\ n)$ time (for any constant $\epsilon \geq 0$).

2. We also present another algorithm on CREW (CRCW) PRAM that uses $O(\frac{nk}{log^2 n})$ $(O(\frac{nk}{log\ n}))$ and requires $O(log^2 n)$ $(O(log\ n))$ time. Note that these algorithms exhibit optimal speed up as k can be at most m. Note also that these algorithms are adaptive in the sense that the processor complexity depends on the number of variables in the pattern.

The rest of the paper is organized as follows. In the next section we sketch the notational preliminaries required for the tree-pattern-matching problem. In Section 3 we give an overview of our algorithms and their detailed description appears in Section 4. Finally, concluding remarks appear in Section 5.

2 Preliminaries

Let V be a set of variables and F be a set of function symbols. We assume that $V \cap F = \Phi$. Each function $f \in F$ has a fixed arity a_f which is a finite non-negative integer. A function symbol with arity 0 is a constant. A *term* is recursively defined as follows.

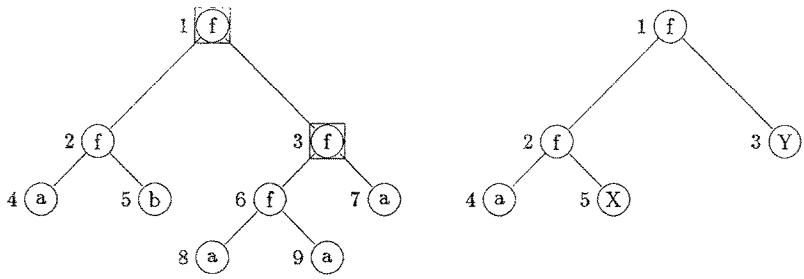

Figure 2: Example 2

- A variable or a constant is a term and,
- if $f \in F$ and $t_1, t_2, \ldots, t_{a_f}$ are terms then so is $f(t_1, t_2, \ldots, t_{a_f})$.

A *ground* term is made up of only function symbols. On the other hand a term containing one or more variables is a *pattern*. A term is represented by a *labeled directed tree* defined as follows.

Definition 1 *A labeled directed tree is a finite directed tree $T(r, N, E)$ such that:*

1. *N is a finite set of nodes and E is the set of edges in the tree.*
2. *r is a special node called root which has indegree 0 and every other node has indegree 1.*
3. *Every node n of the tree T has a unique label that belongs to $V \cup F$.*
4. *if a node n has label $v \in V$, then it has outdegree 0.*
5. *if a node n has label $f \in F$, then it has outdegree a_f, and the edges leaving it are labeled 1,2,..., a_f, respectively.*

The labeled tree representations of a term $s = f(f(a,b), f(f(a,a),a))$ containing symbols from F alone and a term $p = f(f(a,X),Y)$ containing symbols from $F \cup V$ is shown in Figure 2.

Pattern matching on trees is defined as follows. Let s be the tree representing a ground term and p be a pattern tree with k variables.

Definition 2 *p matches s at node x if there exists ground terms $t_1, t_2, \ldots, t_k$ such that the new ground term p' obtained by substituting t_i for the i^{th} variable in p, is identical to the subtree of s rooted at x.*

As an illustration, in Figure 2 the pattern p matches s at node 1 since we can choose $t_1 = f(a,b)$ and $t_2 = f(f(a,a),a)$ as substitutions for X and Y respectively. Similarly, p also matches s at node 3.

In the tree-pattern-matching problem we are required to list all the nodes in s at which a match for p occurs.

3 Overview

In the description and analysis of our algorithms for linear tree pattern matching we will assume that s and p denote the subject and pattern trees respectively. We will also assume that n and m denote the

size of s and p respectively. Throughout this paper we will be using tree s and p in Figure 2 to illustrate our algorithm.

A special case of tree pattern matching is term matching in which we match p with s at its root. Therefore any term-matching algorithm can be used to do tree pattern matching. This is the basis of our first parallel algorithm.

In the second approach we use string matching to do tree pattern matching. Herein, we first transform the subject and pattern trees into a string representation that preserves the structure of the two trees. A succession of string matching is then performed and the results are carefully combined in order to solve the tree-pattern-matching problem.

3.1 Algorithm using Term Matching

The basic idea behind our first approach is to perform pattern matching in trees through a sequence of term matching on the subtrees of s. For every node in s form a subtree rooted at that node consisting of all its descendants. Tree pattern matching now involves term matching p with each of these subtrees.

Certain simple optimizations can be carried out in order to eliminate unnecessary term matching on some of the subtrees of s as these are guaranteed to fail. For instance, term matching p with a subtree t, such that the root labels of p and t are different, is bound to fail.

Dwork et. al [4] and Verma et. al [17] independently proposed the first parallel algorithms for term matching using CREW PRAM. Recently, these algorithms were substantially improved in [14]. This improved algorithm runs on EREW PRAM in $O(n^{\epsilon} log\ n)$ time and uses $O((n+m)^{1-\epsilon})$ $(0 < \epsilon \leq 1)$ processors. Using this algorithm for tree pattern matching would result in a worst-case processor complexity of $O(n^{2-\epsilon})$ and a time complexity of $O(n^{\epsilon} log\ n)$. The worst case processor complexity arises when s is a degenerate tree consisting only of function symbols whose arity is either 1 or 0 and $m \ll n$.

3.2 Algorithm using String Matching

The central idea underlying our second approach requires "linearizing" the pattern and subject trees in such a way that their structural information is preserved. We then perform a sequence of string matches on these linearized structures and combine the outcomes of these matches to solve the tree-pattern-matching problem. The reminder of the paper deals with this approach.

We linearize the tree using its Euler circuit which is obtained as follows. First we replace every edge by a pair of oppositely directed edges. In the resulting structure, the circuit containing all the edges in which each edge occurs exactly once constitutes an Euler circuit. By breaking this circuit at the root we obtain an Euler chain.

In the tree-pattern-matching problem the relative order of the children of a node is crucial. It is therefore important to preserve this ordering in the linearized structure. The construction given earlier for Euler chain does not preserve such an ordering. We therefore use a variant of the above construction in which this ordering is guaranteed to be preserved.

We represent the Euler chain as a string defined recursively as follows.

Definition 3

1. *A string representing the Euler chain of a tree T(r,N,E) such that $N = \{r\}$ and $E = \phi$ is r, i.e., if there is only one node in the tree then the string representing the Euler chain is just that node.*

2. *if T(r,N,E) is a tree and $T_1(r_1, N_1, E_1), T_2(r_2, N_2, E_2), \ldots, T_l(r_l, N_l, E_l)$ are the subtrees rooted at the children $r_1, r_2, \ldots, r_l$ of r respectively, then the string representing the Euler chain of T(r,N,E) is $re_1re_2 \ldots re_lr$ where e_i is the string representing Euler chain of the subtree $T_i(r_i, N_i, E_i)$ rooted at r_i.*

M_1 =

1	0	0	0	0	0	0	8	0	0	0	0	13	0	0	0	0
4	0	0	0	0	0	0	11	0	0	0	0	16	0	0	0	0

M_2 =

1	0	0	0	0	6	7	8	0	0	0	0	13	0	0	16	0
2	0	0	0	0	7	8	9	0	0	0	0	14	0	0	17	0

M_3 =

1	2	0	4	0	6	7	8	9	0	11	0	13	14	0	16	17
1	2	0	4	0	6	7	8	9	0	11	0	13	14	0	16	17

Figure 3:

As an illustration, the strings representing s and p in Figure 2 are 12425213686963731 and 124252131 respectively. Note that our definition preserves the relative ordering of the children of every node.

We use the following key properties of an Euler chain in the design of our algorithm.

1. The leaves of the tree occur only once in the chain.
2. A node x with a_x children occurs $a_x + 1$ times in the chain.
3. The substring in the Euler chain between the first and last occurrence of a node i is the Euler chain of the subtree rooted at i.
4. For a node x with a_x children, the Euler chain of the subtree rooted at its i^{th} child is the substring between the i^{th} and $(i+1)^{\text{th}}$ occurrence of x

In our description matching of two strings refers to matching the labels associated with the nodes in the string.

We now give an overview of our algorithm. Let the strings E_s and E_p be the Euler chains of s and p respectively. Let $\mid E_s \mid$ and $\mid E_p \mid$ denote the size of these strings respectively. We will assume that pattern p has k variables $V_1, V_2, \ldots, V_k$. Since only leaf nodes are associated with the variables and since each leaf appears exactly once in the Euler chain, each of the variables $V_1, V_2, \ldots, V_k$ therefore appear exactly once in E_p.

We first store E_s in an array. Next we split E_p into $k+1$ strings by removing the k variables from it. We denote these strings as $s_1, s_2, \ldots, s_k$. For example, in Figure 2 removing X and Y results in three strings $s_1 = 1242$, $s_2 = 21$, $s_3 = 1$. The corresponding labels associated with the nodes are $fffaf$, ff and f. We then construct tables $M_1, M_2, \ldots, M_k$ where each M_i has $\mid E_s \mid$ entries. If there is a match for s_i in E_s at position j then the j^{th} entry of M_i contains the starting and ending position of this match. If case of no match it is null. For strings s_1, s_2 and s_3, the tables M_1, M_2 and M_3 are shown in Figure 3.

We say that E_p matches a substring of E_s, beginning at the i^{th} position in E_s if there are substrings $e_1, e_2, \ldots, e_k$ of E_s such that the new string $E_p{}'$ obtained by substituting e_i for V_i in E_p is identical to the substring of E_s between positions i and $i+ \mid E_p{}' \mid -1$. In order to ensure that this match corresponds to a tree-pattern match the following conditions must hold.

C_1 : The new string $E_p{}'$ obtained following the substitution must represent an Euler chain of a subtree of s.

C_2 : Each of the e_i's must be Euler chain of a subtree of s.

C_1 can be easily verified since $E_p{}'$ is a Euler chain of the subtree of s rooted at x iff the i^{th} position in E_s is the first occurrence of x in E_s and the $(i+ \mid E_p{}' \mid -1)^{\text{th}}$ position is the last occurrence of x in E_s. The second condition C_2 can be ensured by using only those substrings of E_s whose first and last entries correspond to the first and last occurrence of the same node.

Since s_1 is a prefix of E_p it is obvious that the set of matches for E_p in E_s is a subset of matches for s_1 in E_s. We therefore look for a possible match of E_p in E_s only at the non-null entries in M_1. Let i be one such entry. We then proceed to check whether E_p matches E_s at position i. This is done as follows. $M_1[i][2]$ specifies the position in E_s at which the matching of s_1 ends. Since E_p is $s_1V_1s_2V_2 \ldots s_kV_ks_{k+1}$, the substitution for V_1 must be a substring of E_s that starts from position $M_1[i][2]+1$. This position must correspond to the first occurrence of a node, say x (as only subtrees can be substituted for variables). Otherwise E_p cannot match E_s at i. On the other hand, if this condition is satisfied then the substitution for V_1 will be the substring of E_s which lies between the first and last occurrence of x. Let the last occurrence of x occur at j^{th} position in E_s. We should therefore match s_2 beginning from the $(j+1)^{\text{th}}$ position in E_s. Obviously $M_2[j+1]$ must be a non-null entry. Similarly we compute substitutions for $V_2, V_3, \ldots, V_k$. Upon successful completion of this step E_p will match E_s at position i. We then have to verify that this matched substring satisfies C_1.

We illustrate this using Figure 2 as an example. We start with $M_1[1]$ which is a non-null entry. The match for s_1 ends at the 4^{th} position. Since the 5^{th} position in E_s contains the first occurrence of node 5, substitution for X begins from this position. Since node 5 occurs only once (leaf) it is substituted for X. As we have a non-null entry in M_2 at the 6^{th} position we proceed further. From $M_2[6]$ it is clear that substitution for Y must begin at 8^{th} position. This is the first occurrence of node 3. Substituting the substring between the 8^{th} and 16^{th} position E_s for Y and checking for a match of s_3 at the 17^{th} entry (in M_3) we conclude that E_p matches E_s at its 1^{st} position.

4 Detailed Description

Our implementation of tree pattern matching requires a parallel string-matching algorithm. Both Vishkin[16] and Galil[6] have presented optimal parallel algorithms for string-matching. The algorithm in [16] uses a CRCW PRAM. It requires $O(\frac{n}{log\ n})$ processors and takes $O(log\ n)$ time where n is the size of the input. Galil presents two algorithms in [6]. His first algorithm uses CREW PRAM. It requires $O(\frac{n}{log^2 n})$ processors and takes $O(log^2 n)$ time whereas the second algorithm uses CRCW PRAM having same processor and time complexity as the algorithm in [16]. We will be using the algorithms in [6] for our purpose. Henceforth any reference to parallel string-matching in our paper will refer to these algorithms.

We must have the Euler chains of the pattern and subject trees in an array since the parallel string-matching algorithms require their input to be in this form. To place a linked list in an array we must compute the ranks of each entry in the list and use the ranks as the position for that entry in the array. For this purpose we use the prefix algorithm given in [12]. Informally, given an input $a_1 * a_2 * \ldots * a_n$ with associative operators $*$, the prefix algorithm computes $L_i = a_1 * a_2 * \ldots * a_i$ for each i. The algorithm in [12] uses $O(n)$ processors and takes $O(log\ n)$ time on a EREW PRAM.

Another important result that is used in this paper is due to Brent [2]. This result, referred to as *Brent's theorem*, states that any synchronous parallel algorithm that consists of a total of x elementary operations and takes time t, can be implemented by q processors in time $\lceil \frac{x}{q} \rceil + t$. This implies that a parallel algorithm of processor complexity p_0 and time complexity t_0 can be simulated by $p \leq p_0$ processors in time t such that $pt = p_0t_0$.

We will assume that s and p are represented as labeled directed trees. For convenience we will also assume that these trees are "inverted". In such a tree, a child node points to its parent node. We first transform this inverted tree into its Euler chain which is stored in an array. Tarjan and Vishkin construct an edge-based Euler chain from the adjacency list of a tree [15]. However their construction cannot be readily used to obtain the Euler chain from the inverted tree. We have therefore devised the following construction.

The nodes of the tree are stored in an array T of records. The *label* in a record contains either a function symbol or a variable associated with the node. The *father* field contains a pointer to the

parent of the node and the *edge_label* field contains an integer specifying the node's ordering relative to its sibling. In order to facilitate the construction we also use another field in the record called *tour* which is an array of $a+1$ records where a is the arity of the function symbol associated with that node. The first step in the construction involves forming a linked list using the *tour* fields. Now each record in *tour* field contains two pointers: *node_info* (which is used to point to the record in T that represents the node in the chain) and *tour_info* (which is used to form the linked list in the Euler chain). In addition it also contains three other fields: *subtree, cost* and *type*. In record $T[i]$ the *subtree* field within each entry of *tour* is used for storing the last occurrence of node i in the Euler chain. This information will be needed when we identify substitutions for variables during tree-matching. The *type* field in the j^{th} entry of $T[i].tour$ is used to indicate whether i is a leaf node. In case i is not a leaf then this field specifies whether the j^{th} occurrence of i is the first (last) occurrence of i. The *cost* field is used in the prefix computation in order to compute the rank of each entry in the *tour* field. Finally, each entry of $T[i].tour$ is placed in an array E at a position given by the respective *cost* field. The algorithmic details are given below.

Algorithm GenTour

We assign one processor per node and do the following steps in parallel. Since the root node does not have a father, the processor assigned to the root of the tree is disabled whenever a step in the algorithm requires access to the father of a node. The algorithm often assigns values to pointers which are addresses of variables. For this purpose we assume a function **adr**() that returns the address of a variable.

Step 1 First initialize the $node\_info$ field of each record in *tour*. For processor i representing node i do

1. $T[T[i].father].tour[T[i].edge\_label].node\_info := T[i].father$

Step 2 Now set up the Euler chain using the $tour\_info$ field. For this step we assume the existence of a function **arity**() which returns the arity of a function.

1. $T[T[i].father].tour[T[i].edge\_label].tour\_info := \mathbf{adr}(T[i].tour[1])$
2. $T[i].tour[\mathbf{arity}(T[i].label)+1].tour\_info :=$
   $\mathbf{adr}(T[T[i].father].tour[T[i].edge\_label+1])$

Step 3 Next compute the *type* information. Here we use **leaf** to denote leaf of the tree, **first** to denote the first occurrence of a node in the Euler chain and **last** to denote the last occurrence of a node in the chain. **dummy** is used to denote the other occurrences of a node.

$T[T[i].father].tour[T[i].edge\_label].type := \mathbf{dummy}$
/* first initialize the *type* fields to **dummy** */
if $(\mathbf{arity}(T[i].label) = 0)$ **then**
  $T[i].tour[1].type := \mathbf{leaf}$
else
  $T[i].tour[1].type := \mathbf{first}$
  $T[i].tour[\mathbf{arity}(T[i].label)+1].type := \mathbf{last}$
endif

Step 4 Now place the Euler chain in an array. The position in the array for each entry of *tour* field of $T[i]$ is given by its rank in the linked list representation of the Euler chain. The ranks can be calculated by the prefix algorithm. The ranks replace the contents of *cost* field. Now the *subtree* field of each record in *tour* of $T[i]$ are the contents of the *cost* field of the last record in it.

$T[T[i].father].tour[T[i].edge\_label].subtree :=$
$T[T[i].father].tour[\mathbf{arity}(T[T[i].father].label)+1].cost$

Step 5 Copy each record j in *tour* field of $T[i]$ into the array E at location $j.cost$ as follows.

1. $E[T[T[i].father].tour[T[i].edge\_label + 1].cost] :=$
   $T[T[i].father].tour[T[i].edge\_label]$
2. $E[T[i].tour[1].cost] := T[i].tour[1]$

Complexity: Step 4 of the algorithm takes $O(log\ n)$ time using $O(n)$ processers. All other steps take only constant time. Hence the Euler chain can be constructed in time $O(log\ n)$ using $O(n)$ processors.

Once we have the Euler chain of the subject tree (stored in array E_s) and the pattern tree (stored in array E_p) then the problem of tree-pattern matching can be transformed into one of pattern matching on strings. We accomplish this in two phases. In the first phase we preprocess the pattern and convert it into a set of strings and then match these strings with the Euler chain of the subject to produce the tables $M_1, M_2, \ldots, M_{k+1}$. In the second phase we use the information in these tables to do pattern matching.

In the first phase we convert E_p into a set Σ_p of strings with no variables. Each substring s_i of E_p that lies between successive variables in E_p is an element in Σ_p. In a pattern with k variables we will have $k+1$ strings in Σ_p. Each string in Σ_p is specified by a pair denoting its starting and ending position in E_s. The set Σ_p is organized in a table of $k+1$ entries. $\Sigma_p[i]$ contains the pair (x, y) where x and y are the starting and ending positions of s_i in E_p. Following this step we generate the tables $M_1, M_2, \ldots, M_{k+1}$ as described earlier.

Note that checking for a match in any row of M_1, by assigning one processor to that row, requires sequentially visiting each of the $k+1$ tables. Clearly, this would result in a time complexity of $O(k)$. We can however speed this up using $O(\mid E_s \mid k)$ processors as follows. We first combine pairs of tables M_i and M_{i+1} $(i \in \{1, 3, \ldots, l\}$ $(l = k$ if k is odd else $l = k-1))$. During this step, for every non-null entry in M_i we compute a substitution for V_i (if one exists). Any substitution for V_i relates two entries x and y in tables M_i and M_{i+1} and we replace $M_i[x][2]$ with $M_{i+1}[y][2]$. At the end of this step we are left with $\lceil \frac{k}{2} \rceil$ tables $M_1, M_3, \ldots, M_l$. Repeating this step $O(log\ k)$ times will result in a single table. A non-null entry (x, y) in the final table indicates a match at position x for E_p in E_s.

As an illustration, consider the tables in Figure 3. Combining M_1 and M_2 results in an updated M_1. M_3 is not combined with any other table in this step. The updated M_1 and the unaltered M_3 are shown below.

$M_1 =$

1	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0

$M_3 =$

1	2	0	4	0	6	7	8	9	0	11	0	13	14	0	16	17
1	2	0	4	0	6	7	8	9	0	11	0	13	14	0	16	17

Since there is no substitution for X starting at the 17^{th} position of E_s, the 13^{th} entry in the updated M_1 is null. Combining the updated M_1 with M_3 in the second iteration results in the final table shown below.

$M_1 =$

1	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	16	0	0	0	0	0	0	0	0	0

Observe from the table that E_p matches E_s at its 1^{st} and 8^{th} positions. Since these two positions correspond to the first occurrences of nodes 1 and 3 respectively, pattern tree p matches subject tree s at nodes 1 and 3.

We can reduce the processors required in the above step to $O(\frac{\mid E_s \mid k}{log\ k})$. This is done by first forming $O(\frac{k}{log\ k})$ groups, each of which consists of $log\ k$ tables. The i^{th} group contains tables $M_{i*log\ k+1}$ through $M_{(i+1)*log\ k}$. We assign $\mid E_s \mid$ processors per group to combine all the tables within it. This can be done in $O(log\ k)$ time. The resulting $O(\frac{k}{log\ k})$ tables is then combined as described earlier.

We now give the details of our algorithm. The input consists of two Euler chains E_p and E_s which are two arrays created by algorithm **GenTour**. We will assume a function **variable**() which, given a node label, returns a boolean value true if the the label is a variable and false otherwise. We use the construct **pardo**...**parend** to denote the parallel execution of steps within the construct.

Algorithm TreeMatch

First Phase

Step 1 First preprocess the pattern to create the set of strings and store them in array $\Sigma_p[1..k+1]$. At the end of this step $\Sigma_p[i]$ will contain has a pair of indices into E_p delimiting the i_{th} string. This preprocessing is done as follows. We first assign a cost of 1 to each variable and a cost of 0 to others. Use the prefix algorithm to compute the rank of all the members in the string. The positions preceding and succeeding a variable (whose rank is j) are the ending and starting positions of two consecutive strings j and $j+1$ respectively.

for each entry i **pardo**
  $E_p[i].cost = 0$
  if variable$(E_p[i].node\_info.label)$ **then** $E_p[i].cost = 1$
  endif
parend

Use the prefix algorithm to compute

$E_p[i].cost = \sum_{j=1}^{i} E_p[j]$ $(1 \leq i \leq \mid E_p \mid$
Now complete Σ_p as follows

for each $i > 0$ **pardo**
  if variable$(E_p[i].node\_info.label)$ **then**
    $\Sigma_p[E_p[i].cost][2] = i - 1$
    $\Sigma_p[E_p[i].cost + 1][1] = i + 1$
  endif
parend
$\Sigma_p[1][1] = 1$ /* first string always starts at $i = 1$ */

Step 2 Note that the length of string i is $\Sigma_p[i][2] - \Sigma_p[i][1] + 1$. Now use the parallel string matching algorithm to construct the tables $M_1, M_2, \ldots, M_{k+1}$ in parallel. This concludes the first phase of our algorithm.

Second Phase Now combine the tables to produce the result of tree-pattern matching in M_1. We first combine every $log\ k$ tables in parallel to produce $O(\frac{k}{log\ k})$ tables. After that in $O(log\ k)$ time we combine these tables into M_1. For simplicity let us assume that $k+1$ is divisible by $\lceil log\ k \rceil$ and $r = \frac{k+1}{\lceil log\ k \rceil}$

Step 1 This step combines each of the $log\ k$ tables

for each $i \in \{1, 2, \ldots, r\}$ **pardo**
  for each $j \in \{1, 2, \ldots, \mid E_s \mid\}$ **pardo**
    for $l = 1$ **to** $\lceil log\ k \rceil$ **do**
      if $(M_i[j][1] \neq 0)$ **then**
        if $(E_s[M_i[j][2]+1].type = \textbf{first} \vee E_s[M_i[j][2]+1].type = \textbf{leaf})$ **then**
          if $(M_{i+l}[E_s[M_i[j][2]+1].subtree + 1][1] \neq 0)$ **then**
            $M_i[j][2] = M_{i+l}[E_s[M_i[j][2]+1].subtree + 1][2]$
          else $M_i[j][1] = M_i[j][2] = 0$
          endif
        else $M_i[j][1] = M_i[j][2] = 0$
        endif
      endif

```
        endfor
      parend
    parend
```

Step 2 Now tables $M_{i*log\ k}(i \in \{1,2,\ldots,r\})$ contain the results of combining M_i through $M_{(i+1)*log\ k-1}$. However for simplicity of notation we assume that the results of above step is available in tables M_1 through M_r.

```
for each i ∈ {1,2,...,r} pardo
  for each j ∈ {1,2,...,| E_s |} pardo
    for l = 1 to log r do
      if (i + 2^l ≤ r) then
        if (M_i[j][1] ≠ 0) then
          if (E_s[M_i[j][2]+1].type = first ∨ E_s[M_i[j][2]+1].type = leaf) then
            if (M_{i+2^l}[E_s[M_i[j][2]+1].subtree+1][1] ≠ 0) then
              M_i[j][2] = M_{i+2^l}[E_s[M_i[j][2]+1].subtree+1][2]
            else M_i[j][1] = M_i[j][2] = 0
            endif
          else M_i[j][1] = M_i[j][2] = 0
          endif
        endif
      endif
    endfor
  parend
parend
```

Step 3 Finally we check whether C_1 is satisfied

```
for each i ∈ 1,2,...,| E_s |
  if(M_1[i][1] ≠ 0) then
    if(E_s[M_1[i][1]].type ≠ first ∨ E_s[M_1[i][2]].type ≠ last) then
      M_1[i][1] = M_1[i][2] = 0
    endif
  endif
parend
```

Now the non-null entries in M_1 give the position at which E_p matches E_s. If i is one such entry then $E_s[M_1[i][1]].node\_info$ is the node in s at which there is a match for$p$.

Complexity: Step 1 of the first phase can be computed in $O(log\ n)$ time using $O(n)$ processors. The second step involves string matching of $O(k)$ strings with E_s. Since $\mid E_s \mid$ is $O(n)$ this step can be carried out using $O(\frac{nk}{log\ n})$ processors in time $O(log\ n)$. As described earlier the second phase of our algorithm can be carried out with $O(\frac{nk}{log\ k})$ processors in time $O(log\ k)$. Since $log\ n > log\ k$, by Brent's theorem this phase can also be carried out in time $O(log\ n)$ with $O(\frac{nk}{log\ n})$ processors. The parallel string-matching algorithm used in Step 2 of the first phase requires CRCW PRAM. All the other steps can be implemented on CREW PRAM.

The parallel string matching algorithm can also be implemented on CREW PRAM with a processor complexity of $O(\frac{n}{log^2 n})$ and a time complexity of $O(log^2 n)$. So by Brent's theorem our algorithm takes $O(log^2 n)$ time on CREW PRAM and uses $O(\frac{nk}{log^2 n})$ processors.

Observe that our algorithm achieves optimal speedup as its worst case processor-time product is $O(nm)$ since k can be at most m.

5 Conclusion

In this paper we described parallel algorithms for linear tree pattern matching on the PRAM model. To the best of our knowledge these are the first known parallel algorithms for this problem on this model. Our algorithms on CRCW and CREW models exhibit optimal speedup, in the sense that the processor-time product matches the worst-case time complexity of the fastest known sequential algorithm in [8]. It is straightforward to extend our algorithm for matching several patterns with a single subject. In particular if q is the total number of patterns and $k_1, k_2, \ldots, k_q$ are the number of variables in pattern $p_1, p_2, \ldots, p_q$ respectively, then our algorithm would require $O(\frac{(q+\sum_{i=1}^{q} k_i)n}{log^2 n})$ processors and take $O(log^2 n)$ time on CREW model. On the CRCW model the algorithm would require $O(\frac{(q+\sum_{i=1}^{q} k_i)n}{log\ n})$ processors and take $O(log\ n)$ time. This algorithm also achieves optimal speedup as its processor-time product matches the worst-case time complexity of the fastest known sequential algorithm in [8] for matching multiple patterns with a single subject.

References

[1] A.V. Aho and M. Ganapathy, Efficient Tree Pattern Matching: An Aid to Code Generation, Proceedings of the Eleventh ACM Symposium on Principles of Programming Languages, 1984 pp. 334-340.

[2] R. P. Brent, The Parallel Evaluation of General Arithmetic Expressions, JACM 21,2 (1974), pp. 201-206.

[3] G. Collins, The SAC-1 System: An introduction and Survey, Proceedings of the Second ACM Conference on Symbolic and Algebraic Manipulation, 1971, pp. 144-152.

[4] C. Dwork, P. Kanellakis and L. Stockmeyer, Parallel Algorithm for Term Matching, Eighth Conference on Automated Deduction, Oxford University, July 1986.

[5] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, Proceedings of the Tenth ACM Symposium of Theory of Computing, 1978 pp. 114-118.

[6] Z. Galil, Optimal Parallel Algorithms, VLSI: Algorithms and Architectures, P. Bertolazzi and F. Luccio (editors), Elsevier Science Publishers B.V. (North Holland), 1985 pp. 3-10.

[7] J. Guttag, E. Horowitz and D.R. Musser, Abstract Data Types and Software Validation, CACM 21, 12, December 1978 pp. 1048-1064.

[8] C.M. Hoffmann and M.J. O'Donnell, Pattern Matching in Trees, JACM 29, 1, 1982 pp. 68-95.

[9] C.M. Hoffmann and M.J. O'Donnell, An Interpreter Generator using Tree Pattern Matching, Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, 1979 pp. 169-179.

[10] R. Karp, R.E. Miller and A. Rosenberg, Rapid Identification of Repeated Patterns in Strings, Trees and Arrays, Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, Denver, Colorado 1972, pp. 125-136.

[11] H. Kron, Tree Templates and Subtree Transformational Grammars, Ph.D Dissertation, University of California, Santa Cruz, 1975.

[12] C. P. Kruskal, L. Rudolph and M. Snir, Efficient Parallel Algorithms for Graph Problems, Proceedings of the 1986 International Conference on Parallel Processing, pp. 180-185.

[13] P. W. Purdom, Jr. and C. A. Brown, Fast Many-to-One Matching Algorithms, Proceedings of the First International Conference on Rewriting Techniques and Applications, 1985, Dijon, France.

[14] R. Ramesh, R. M. Verma, T. Krishnaprasad and I.V. Ramakrishnan, Term Matching on Parallel Computers, SUNY Stony Brook Computer Science Dept. Tech Report No. 86/20, November 1986 (To appear in the Proceedings of Fourteenth ICALP).

[15] R. E. Tarjan and U. Vishkin, Finding Bi-Connected Components and Computing Tree Functions in Logarithmic Parallel Time, Proceedings of the Twentyfifth IEEE Symposium on Foundations of Computer Science, 1984, pp. 12-20.

[16] U. Vishkin, Optimal Parallel Pattern Matching in Strings, Twelfth ICALP, Lecture Notes in Computer Science 194, Springer-Verlag, 1985, pp. 497-508.

[17] R. M. Verma, T. Krishnaprasad, I.V. Ramakrishnan, An Efficient Parallel Algorithm for Term Matching, Sixth International Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture notes in Computer Science, Springer-Verlag, December 1986, pp. 504-518

[18] R. Cole and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, Proceedings of the Eithteenth Annual ACM Symposium on Theory of Computing, 1986, pp. 206-219.