Title	An optimal EREW parallel algorithm for parenthesis matching
Other Contributor(s)	University of Hong Kong. Dept. of Computer Science
Author(s)	Chin, Yo-lun, Francis; Lam, Tak-wah; Tsang, Wai-wan
Citation	
Issued Date	1989
URL	http://hdl.handle.net/10722/54876
Rights	Creative Commons: Attribution 3.0 Hong Kong License

COMPUTER SCIENCE PUBLICATION

An Optimal EREW Parallel Algorithm for

Parenthesis Matching

W.W. Tsang, T.W. Lam & F.Y.L. Chin

Technical Report TR-89-01

January 1989

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF HONG KONG
POKFULAM ROAD
HONG KONG

An Optimal EREW Parallel Algorithm for Parenthesis Matching

Wai Wan Tsang, Tak W. Lam and Francis Y.L. Chin*
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong

January 16, 1989

Abstract

Parenthesis matching is an important step in the construction of computation tree form and parsing. A new parallel algorithm is presented for matching a string of n parentheses in $O(\log n)$ time using $n/\log n$ processors on an exclusive-read, exclusive-write parallel random-access machine (EREW PRAM). The previously best known method requires $O(\log^2 n)$ time and $n/\log n$ processors on an EREW PRAM or a concurrent-read, exclusive-write parallel random-access machine (CREW PRAM) for $O(\log n)$ time complexity.

Keywords: Parallel Optimal Algorithm, EREW PRAM, Parenthesis Matching, Pipelining, Computation Tree.

1 Introduction

Let $x_1, x_2, ..., x_n$ be a string of parentheses. The parenthesis matching problem is to find all pairs of matched parentheses if the string is well-formed [BaVi] otherwise output an error message. It is easy to construct an $O(n)$ time sequential algorithm for solving the problem using a stack. In this paper, we consider solving this problem on an exclusive-read, exclusive-write parallel random-access machine (EREW PRAM) [FoWy, Gold].

Parenthesis matching is a classical problem in parsing. This matching process together with the application of Knuth's parenthesis-insertion technique [Knu] are important steps in the generation of computation tree forms for arithmetic expressions [BaVi]. The parallel algorithm for the generation of computation tree forms,

*electronic-mail address: hkucs!tsang@uunet.uu.net, hkucs!twlam@uunet.uu.net, hkucs!chm@uunet.uu.net

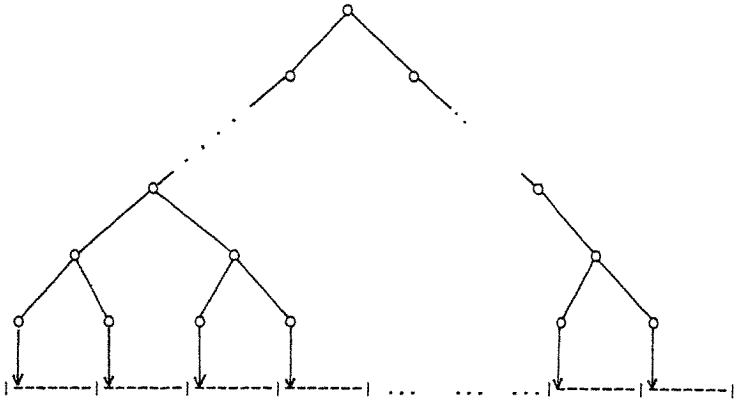

Figure 1: Organization of Processors

which relies on parenthesis matching, was first studied by Dekel and Sahni [DeSa]. They adapted the known sequential algorithm and emulated the stack in parallel [Reif]. The resulting algorithm runs in $O(\log^2 n)$ time using n processors, or $O(\log n)$ time with $n^2/\log n$ processors, on an EREW PRAM. Bar-on and Vishkin [BaVi] used a binary tree structure to solve this problem and obtained an $O(\log n)$ time and $n/\log n$ processors algorithm on a concurrent-read and exclusive-write parallel random-access machine (CREW PRAM). Their algorithm can be run on an EREW PRAM by simulation [Eck,Vis]. It takes $O(\log^2 n)$ time using $n/\log n$ processors and is an improvement over Dekel and Sahni's result on the number of processors used.

In our algorithm, processors are organized in the form of a full binary tree. Each leaf processor is assigned to a small segment of the parenthesis string as shown in Figure 1. The entire algorithm consists of three stages. The first stage is exactly the same as the first part of the Bar-on and Vishkin's algorithm. The parentheses in each segment are matched sequentially by the leaf processor assigned. In the second stage, information of the remaining unmatched parentheses in each segment is passed upward from the leaves to the root of the processor tree. The third stage starts when this information arrives at the root. A unique identifier is allocated to each matched pair of parentheses. This stage involves passing the allocation information downward from the root to the leaves. Eventually, each parenthesis receives an identifier.

With the identifier information, the matching of parentheses can be easily done in two rounds using two arrays L and R. In the first round, set $L[h] = i$ if x_i is a left parenthesis and the identifier so allocated is h; set $R[h] = i$ if x_i is a right parenthesis. In the second round, an array M is constructed such that $M[i] = j$ if

and only if x_i matched with x_j in the following way.

for each h allocated
$M[L[h]] = R[h]$;
$M[R[h]] = L[h]$
end-for

Our algorithm requires $n/\log n$ processors but performs neither a concurrent-read nor concurrent-write operations on a PRAM. By pipelining the operations in the third stage, the entire process can be done in $O(\log n)$ time. This is a significant improvement over the previous best method [BaVi] as the latter requires a CREW PRAM for similar time and processor complexity.

A detailed description of the algorithm is given in next section followed by its proof of correctness in Section 3. Section 4 gives the analysis of the time and processor complexity. Some general remarks are presented in Section 5.

2 The Algorithm

Our parenthesis matching algorithm can be divided into 3 stages: the initial stage, the upward stage and the downward stage.

2.1 Initial stage

The input string is partitioned into $n/(2\log n)$ successive segments of length $2\log n$ each. A leaf processor of the tree mentioned in Section 1 is assigned to each segment. The matching of parentheses within a segment is done by the processor assigned with one pass on the segment using a stack. The remaining unmatched parentheses in each segment will be of the form $))\cdots))((\cdots($, i.e., a sequence of right parentheses followed by a sequence of left parentheses. Since the matched parentheses in each segment can be handled in this stage independently, without loss of generality, we shall assume from now on that all segments contain only the unmatched left and right parentheses.

2.2 Upward Stage

Assuming that $n/\log n$ is a power of 2, the tree shown in Figure 1 consists of $n/\log n - 1$ processors and $n/(2\log n)$ of them are leaves. For convenience, in the following whenever without confusion, we shall use node and processor interchangeably. Moreover, for any node v on the tree, we define the *substring* of v (or v's *substring*) as the string formed by concatenating the segments assigned to the

leaves of the subtree rooted at node v (Figure 1) and refer the parentheses in the substring of v as *parentheses at* v. In this stage, each node v computes and passes a triple, $\langle m, r, l \rangle$, as a packet to its parent, where m, r and l are the number of pairs of matched parentheses and the numbers of unmatched right and left parentheses in the substring of node v, respectively. Note that the triple computed at a leaf node is of the form $\langle 0, r, l \rangle$. The triple information will be used later in allocating identifiers in the third stage.

Throughout this paper, we assume that node v_j and node v_k are the left and right children of node v_i in the processor tree. Let $\langle m_j, r_j, l_j \rangle$ and $\langle m_k, r_k, l_k \rangle$ be the triples computed at v_j and v_k, respectively. First, let us consider the case when $l_j \leq r_k$. Then all unmatched left parentheses at node v_j match with the leftmost unmatched right parentheses at node v_k. In fact, the remaining unmatched right parentheses at node v_k and all unmatched right parentheses at node v_j will contribute to the unmatched right parentheses at node v_i. In general, the triple of a non-leaf node v_i is computed from the triples of its children with the following formulas:

$$
\begin{aligned}
m_i &= m_j + m_k + \min(l_j, r_k) \\
r_i &= r_j + r_k - \min(l_j, r_k) \\
l_i &= l_j + l_k - \min(l_j, r_k)
\end{aligned}
$$

It can be shown that the value m in the triple at the root is the number of pairs of matched parentheses in the input string, and that r and l are zero if and only if the input string is well-formed.

Communications between a parent node and its two children are carried by sending packets along tree edges. When we say that a packet is sent from one processor and received by another, we mean that a processor writes the information into a shared memory location which is read by the other in a later step. Each step takes constant time, say one time unit, and a step is sub-divided into phases: reception phase, execution phase and transmission phase. During each step, only one packet can be received and transmitted. Thus, if a packet is sent at time t by one processor, it can only be received by another processor at time $t + 1$ or later.

2.3 Downward stage

In this stage, each matched pair of parentheses in the input string will be assigned with a distinct identifer selected from the range $[1..n/2]$.

At any node v, there are two types of matched parentheses:

1. pairs of parentheses matched within the substring of either one of its children, and
2. pairs of parentheses matched across substrings of its two children.

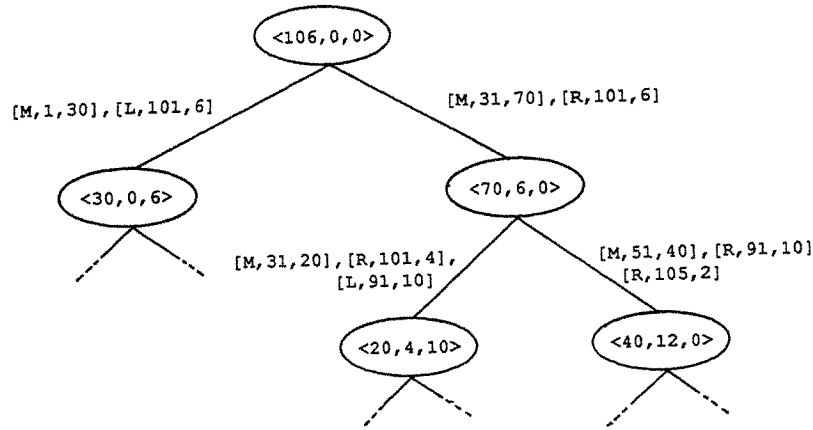

Figure 2: A Simple Example

Starting from the root, every node allocates an identifier, within the range specified by its parent, to each pair of its type 2 parentheses. This allocation information, together with identifier subranges reserved for the matched pairs in its children's substrings (i.e., of type 1), are passed downward as a sequence of packets. Eventually, a leaf assigns identifiers to the parentheses in its attached segment according to the contents and order of packets it receives. Each non-leaf node, in addition to allocating identifiers to its type 2 parentheses, has to relay, and sometimes split, the allocation packets received from its parent.

Figure 2 gives an example of the configuration of the top part of the processor tree right after the upward stage. When the downward stage starts, a range of identifiers, $1 \cdots 106$, which can be treated as indices in arrays R and L, is reserved by the root for all pairs of matched parentheses. According to the triples previously received from its children, the root divides this range into three parts: $1 \cdots 30$ for the matched pairs in its left child's substring, $31 \cdots 100$ for the matched pairs of its right child's and $101 \cdots 106$ for the parentheses which match across the substrings of its two children. It then informs its children of the allocation by passing two packets, $[M, 1, 30]$ and $[L, 101, 6]$, to its left child and two packets, $[M, 31, 70]$ and $[R, 101, 6]$, to its right child. Note that the first component of each packet specifies the type of parentheses, while the second and the third components specify the starting location and the number of identifiers allocated. In fact, the first component of each packet is redundant if a proper protocol is used.

When the right child receives the packets, it further divides the range $31 \cdots 100$ into three parts, $31 \cdots 50$, $51 \cdots 90$, and $91 \cdots 100$, in the same way as before. In addition, since four of its six unmatched right parentheses come from the left child and the others from the right, it divides the range, $101 \cdots 106$, into two parts: $101 \cdots 104$

and $105 \cdots 106$. It then sends packets, $[M, 31, 20]$, $[R, 101, 4]$ and $[L, 91, 10]$ to its left child and packets, $[M, 51, 40]$, $[R, 91, 10]$ and $[R, 105, 2]$, to its right.

The packets are split and passed downward and eventually arrive at the leaves. The leaf processors then store the index of each parenthesis into the slot, specified by the identifier received, of the array R or L. Note that the packets should be arrived in the same order of the corresponding unmatched parentheses in the substring. In our algorithm, we adopt the following ordering of the unmatched parentheses in a substring,

$$\begin{array}{cccccccccccc}) & \cdot\cdot &) & \cdot\cdot &) & \cdots &) & (& \cdots & (& \cdot\cdot & (\\ 1 & & 2 & & 3 & \cdots & x & y & \cdots & 3 & & 2 \quad 1 \end{array}$$

The first R-packet contains the allocation information of the leftmost unmatched right parentheses, while the first L-packet contains the rightmost unmatched left ones. The order of other packets can be deduced in a similar way.

The followings describe in general how the outgoing packets are computed from the incoming sequence of packets and the triples of the children nodes. The packets are denoted with capital letters and their order of arrival is indicated in the listing from left to right. Let $\langle m_i, r_i, l_i \rangle$ be the triple of node v_i and $M_i, R_1, R_2, \cdots R_x, L_1, \cdots L_y$ be incoming packets to node v_i, where $M_i = [M, s(M_i), m_i]$, $R_h = [R, s(R_h), n(R_h)]$ for $1 \leq h \leq x$ and $L_h = [L, s(L_h), n(L_h)]$ for $1 \leq h \leq y$. Our algorithm ensures that $n(R_1) + n(R_2) + \cdots + n(R_x) = r_i$ and $n(L_1) + n(L_2) + \cdots + n(L_y) = l_i$. Since there are two types of matched parentheses at v_i, four packets have to be derived from M_i and two for each child, v_j or v_k:

$$\begin{aligned} M_j &= [M, s(M_j), m_j] \\ M_k &= [M, s(M_k), m_k] \\ L_m &= [L, s(L_m), n(L_m)] \\ R_m &= [R, s(R_m), n(R_m)] \end{aligned}$$

where

$$\begin{aligned} s(M_j) &= s(M_i) \\ s(M_k) &= s(M_i) + m_j \\ s(L_m) &= s(R_m) = s(M_i) + m_j + m_k \\ n(L_m) &= n(R_m) = \min(l_j, r_k) \end{aligned}$$

Note that packets L_m and R_m have assigned the same identifiers as they correspond to the same pairs of matched parentheses.

Without loss of generality, let us assume that $l_j \leq r_k$. The r_i right parentheses at node v_i are contributed from node v_j and node v_k. Thus, the sequence of packets, $R_1, \cdots, R_w, \cdots, R_x$, have to be divided into two subsequences: $R_1, \cdots, R_{w'}$ for node v_j and $R_{w''}, \cdots, R_x$ for node v_k, where $R_{w'}$ and $R_{w''}$ are split from R_w such that

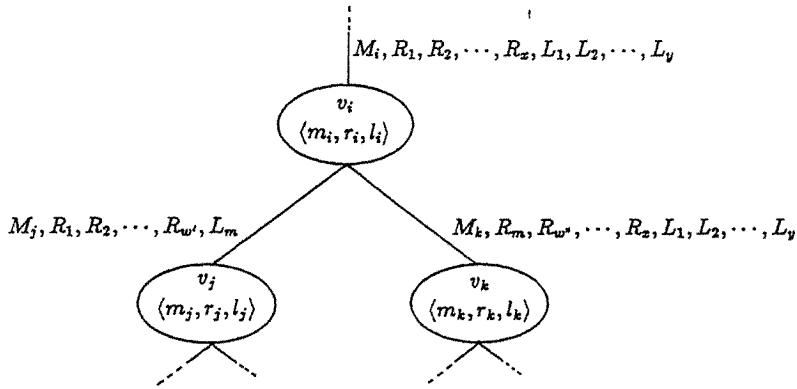

Figure 3: Packets flow through node v_i when $l_j \leq r_k$.

the total number of identifiers specified in $R_1, \cdots, R_{w'}$ equals r_j, i.e., $n(R_1) + \cdots + n(R_{w-1}) + n(R_{w'}) = r_j$. After the leftmost unmatched right parentheses of node v_k have matched all the unmatched left parentheses at node v_j, the remaining unmatched right parentheses then contribute to the rightmost unmatched right parentheses at node v_i. The packets corresponding to the right parentheses received by node v_k should be in the order of $R_m R_{w''} \cdots R_x$ and $r_k = n(R_m) + n(R_{w''}) + n(R_{w+1}) + \cdots + n(R_x)$. (See Figure 3.)

Note that the number of outgoing packets at each node is at most four more than the number of incoming packets. Since the splitting and passing of a packet are independent of the trailing packets, the sending and receiving of packets at each node can be overlapped to speed up the whole process. In most cases, a packet received by a node is relayed to one of its sons as soon as it arrives; this refers to packets: $R_1, \cdots, R_{w-1}, R_{w+1}, \cdots, R_z, L_1, \cdots, L_y$. For packet M_i, node v_i first sends packets M_j and M_k, and L_m and R_m later, to its children. As for packet R_w, two packets $R_{w'}$ and $R_{w''}$ are sent.

Similarly, for the case when $l_j > r_k$, the packets, $L_1, L_2, \cdots, L_w, \cdots, L_y$, have to be divided into two parts: $L_1, \cdots, L_{w'}$ and $L_{w''}, \cdots, L_y$ such that $n(L_1) + \cdots + n(L_{w'}) = l_k$. The packets received by node v_j and node v_k will be in the order of $M_j, R_1, \cdots, R_x, L_m, L_{w''}, \cdots, L_y$ and $M_k, R_m, L_1, \cdots, L_{w'}$, respectively.

The detailed algorithm of the downward stage at node v_i can be described as follows:

Initially, assume that local variables $sum_l = sum_r = 0$ and node v_i knows the triples of its children, i.e., $\langle m_j, r_j, l_j \rangle$ and $\langle m_k, r_k, l_k \rangle$. Let P be the next packet arrived.

CASE

P is of type M:

compute M_j, M_k, L_m and R_m;

send M_j and M_k to node v_j and node v_k, respectively;

send R_m to node v_k

P is of type R and $l_j \leq r_k$:

$sum_r \leftarrow sum_r + n(P)$;

CASE

$sum_r \leq r_j$: relay P to node v_j;

$sum_r - n(P) < r_j < sum_r$:

split P into $R_{w'}$ and $R_{w''}$;

send $R_{w'}$ and L_m to node v_j;

send $R_{w''}$ to node v_k

$r_j \leq sum_r - n(P)$: relay P to node v_k

END-CASE

P is of type R and $l_j > r_k$: relay P to node v_j

P is of type L and $l_j > r_k$:

$sum_l \leftarrow sum_l + n(P)$;

CASE

$sum_l \leq l_k$: relay P to node v_k;

$sum_l - n(P) < l_k < sum_l$:

split P into $L_{w'}$ and $L_{w''}$;

send $L_{w'}$ to node v_k

send L_m and $L_{w''}$ to node v_j;

$l_k \leq sum_l - n(P)$: relay P to node v_j

END-CASE

P is of type L and $l_j \leq r_k$: relay P to node v_k

END-CASE

Eventually, each leaf node v_i will receive a sequence of (at most $\log n+1$) packets $M_i, R_1, \cdots, R_x, L_1, \cdots, L_y$ from its parent. The range specified in M_i is zero and the total number of identifiers allocated in the R-packets (L-packets) should be equal to the number of unmatched right (left) parentheses in the substring of v_i, i.e., r_i (l_i). The identifiers specified by the incoming packets $R_1, \cdots, R_x$ are ordered as follows: $s(R_1), \cdots, s(R_1)+n(R_1)-1, \cdots, s(R_x), \cdots, s(R_x)+n(R_x)-1$. This sequence of identifiers are allocated one by one to the unmatched right parentheses in the segment from left to right. Similarly, identifiers specified by L-packets received are assigned to the left unmatched parentheses from right to left.

3 Proof of Correctness

It is obvious that our algorithm rejects all strings of parentheses which are not well-formed. For those well-formed strings, the correctness of our algorithm follows from corollaries 3, 6 and 8 in the following.

Definition 1 For any node v_i in the binary tree,

1. Let $match_l(v_i)$ $\{match_r(v_i)\}$ be the sequence of left {right} parentheses at v_i, whose matching right {left} parentheses also lie within the substring of v_i.
2. Let $unmatch_l(v_i)$ $\{unmatch_r(v_i)\}$ be the sequence of left {right} parentheses at v_i, whose matching left {right} parentheses do *not* lie within the substring of v_i.
3. If v_i is not a leaf node, let v_j and v_k be its left and right child, respectively. Define $just\text{-}match_l(v_i) = match_l(v_i) - (match_l(v_j) \cup match_l(v_k))$[1]; $just\text{-}match_r(v_i) = match_r(v_i) - (match_r(v_j) \cup match_r(v_k))$.

Fact Consider any non-leaf node v_i. Let v_j and v_k be its left and right child, respectively. Then

1. $just\text{-}match_l(v_i) \subseteq unmatch_l(v_j)$; $just\text{-}match_r(v_i) \subseteq unmatch_r(v_k)$.
2. $|just\text{-}match_l(v_i)| = |just\text{-}match_r(v_i)| = \min(|unmatch_l(v_j)|, |unmatch_r(v_k)|)$.
3. The p^{th} rightmost left parenthesis in $just\text{-}match_l(v_i)$, actually the p^{th} rightmost parenthesis in $unmatch_l(v_j)$, is *matched* with the p^{th} leftmost right parenthesis in $just\text{-}match_r(v_i)$, which is also the p^{th} leftmost parenthesis in $unmatch_r(v_k)$.
4. $unmatch_l(v_i) = (unmatch_l(v_j) - just\text{-}match_l(v_i)) \cdot unmatch_l(v_k)$;[2]
   $unmatch_r(v_i) = unmatch_r(v_j) \cdot (unmatch_r(v_k) - just\text{-}match_r(v_i))$.

Lemma 2 For any node v_i in the tree, let $\langle m_i, r_i, l_i \rangle$ be the triple computed at v_i during the *upward* stage, and $M_i, R_1, \cdots, R_x, L_1, \cdots, L_y$ be the packets received by v_i during the *downward* stage. Then

1. $\sum_{1 \le h \le x} n(R_h) = r_i = |unmatch_r(v_i)|$,
2. $\sum_{1 \le h \le y} n(L_h) = l_i = |unmatch_l(v_i)|$, and
3. $n(M_i) = m_i = |match_l(v_i)| = |match_r(v_i)|$.

[1] For any sequences S_1 and S_2, $S_1 - S_2$ is a sequence formed by deleting all common elements between S_1 and S_2 from S_1.

[2] $\cdot$ is the *concatenation* operator.

Proof: (Sketch) To prove $\sum_{1 \le h \le x} n(R_h) = r_i$, we use induction on the depth of v_i, i.e., starting from the root down to leaf nodes. For $r_i = |unmatch_r(v_i)|$, induction can be used again, but from leaf nodes to the root. The other equalities are proved in a similar way. □

Corollary 3 At the end of the *downward* stage, each leaf node has enough identifiers to assign to parentheses in its own substring. In other words, each parenthesis will be assigned with an identifier.

Definition 4 For any sequence of packets, $P_1, \cdots, P_x$, let $\langle P_1, \cdots, P_x \rangle$ denote the ordered-list of corresponding identifiers, i.e., $(s(P_1), s(P_1)+1, \cdots, s(P_1)+n(P_1)-1, s(P_2), \cdots, s(P_x), \cdots, s(P_x)+n(P_x)-1)$.

Lemma 5 Let P_i, P_j be any two distinct packets of the same type, received by some node(s) of the same depth in the tree. Then $\langle P_i \rangle$ and $\langle P_j \rangle$ do not contain any identifer in common.

Proof: (Sketch) The lemma can be proved by induction on the depth of the tree with the root as the base case. □

Corollary 6 At the end of the *downward* stage, any two distinct *left* (or *right*) parentheses are assigned with distinct identifiers.

Lemma 7 Consider any node v_i in the tree, let $L_1, \cdots, L_y$ and $R_1, \cdots, R_x$ be the sequences of L-packets and R-packets received by v_i, respectively. Then for any $1 \le p \le |unmatch_l(v_i)|$ $\{|unmatch_r(v_i)|\}$, the p^{th} rightmost left {leftmost right} parenthesis in $unmatch_l(v_i)$ $\{unmatch_r(v_i)\}$ will be assigned with the p^{th} identifier of the list $\langle L_1, \cdots, L_y \rangle$ $\{\langle R_1, \cdots, R_x \rangle\}$.

Proof: We prove lemma 7 by induction on the depth of node v_i.

By corollary 3 and the construction of the algorithm, the base case, in which v_i is a leaf node (i.e., depth of $v_i = 0$), is obvious. Now assume the lemma is true for any node of depth $\le d$, for some integer $d \ge 0$. Consider the case in which the depth of v_i is $d+1$. Let v_j and v_k be the left and right child of v_i, respectively. Let $L_1, \cdots, L_y$ and $R_1, \cdots, R_x$ be the sequences of L-packets and R-packets received by v_i.

Here we only prove the case when $r_k \le l_j$. The other case is similar. According to our algorithm, the packets received by v_j and v_k are $M_j, R_1, \cdots, R_x, L_m, L_{w''}, \cdots, L_y$ and $M_k, R_m, L_1, \cdots, L_{w'}$, respectively. Note that $n(L_m) = \min(l_j, r_k) = \min(|unmatch_l(v_j)|, |unmatch_r(v_k)|) = |just\text{-}match_l(v_i)|$. Also, $\langle L_1, \cdots, L_y \rangle = \langle L_1, \cdots, L_{w'} \rangle \cdot \langle L_{w''}, \cdots, L_y \rangle$.

By fact 4, for any $1 \leq p \leq |unmatch_l(v_k)|$, the p^{th} rightmost left parenthesis in $unmatch_l(v_i)$ is actually the p^{th} rightmost left parenthesis in $unmatch_l(v_k)$. By induction hypothesis, the p^{th} rightmost left parenthesis in $unmatch_l(v_k)$ is assigned with the p^{th} identifier in $\langle L_1, \cdots, L_{w'} \rangle$. Thus, the p^{th} rightmost left parenthesis in $unmatch_l(v_i)$ is assigned with p^{th} identifier in $\langle L_1, \cdots, L_{w'} \rangle$, which is also the p^{th} identifier in $\langle L_1, \cdots, L_y \rangle$.

For any $|unmatch_l(v_k)| < p \leq |unmatch_l(v_i)|$, the p^{th} rightmost left parenthesis in $unmatch_l(v_i)$ is the $(p - |unmatch_l(v_k)| + |just\text{-}match_l(v_i)|)^{th}$ rightmost left parenthesis in $unmatch_l(v_j)$. Then by induction hypothesis, the latter is assigned with the $(p - |unmatch_l(v_k)| + |just\text{-}match_l(v_i)|)^{th}$ identifier in $\langle L_m, L_{w''}, \cdots, L_y \rangle$. Since $n(L_m) = |just\text{-}match_l(v_i)|$ and by corollary 3, $\sum_{1 \leq h \leq w'} n(L_h) = |unmatch_l(v_k)|$, the $(p - |unmatch_l(v_k)| + |just\text{-}match_l(v_i)|)^{th}$ identifier in $\langle L_m, L_{w''}, \cdots, L_y \rangle$ is also the $(p - |unmatch_l(v_k)|)^{th}$ identifier in $\langle L_{w''}, \cdots, L_y \rangle$, or equivalently the p^{th} identifier in $\langle L_1, \cdots, L_{w'} \rangle \cdot \langle L_{w''}, \cdots, L_y \rangle = \langle L_1, \cdots, L_y \rangle$.

Since $|unmatch_r(v_i)| = r_k \leq l_j = |unmatch_l(v_j)|$, so $|just\text{-}match_r(v_i)| = \min(|unmatch_l(v_j)|, |unmatch_r(v_k)|) = |unmatch_r(v_k)|$. Moreover, with fact 1, we can further deduce that $just\text{-}match_r(v_i) = unmatch_r(v_k)$. Recall fact 4 that $unmatch_r(v_i) = unmatch_r(v_j) \cdot (unmatch_r(v_k) - just\text{-}match_r(v_i))$. Thus, $unmatch_r(v_i) = unmatch_r(v_j)$.

For any $1 \leq p \leq |unmatch_r(v_i)|$, the p^{th} leftmost right parenthesis in $unmatch_r(v_i)$ is exactly p^{th} leftmost right parenthesis in $unmatch_r(v_j)$. By induction hypothesis, this parenthesis is assigned with the p^{th} identifier in $\langle R_1, \cdots, R_x \rangle$.

This completes the induction step and hence proves lemma 7. □

Corollary 8 At the end of the *downward* stage, any pair of matched parentheses, (x_a, x_b), will be assigned with the same identifier.

Proof: Let v_i be the node of lowest depth, whose substring containing both x_a and x_b. In other words, x_a is in $just\text{-}match_l(v_i)$ and x_b is in $just\text{-}match_r(v_i)$. From fact 3, we can deduce that the relative position of x_a in $just\text{-}match_l(v_i)$ and x_b in $just\text{-}match_r(v_i)$ is the same, i.e., for some $1 \leq p \leq |just\text{-}match_l(v_i)|$, x_a $\{x_b\}$ is the p^{th} rightmost $\{$leftmost$\}$ parenthesis in $just\text{-}match_l(v_i)$ $\{just\text{-}match_r(v_i)\}$. Obviously, v_i cannot be a leaf node. Let v_j and v_k be its left and right child, respectively.

According to the algorithm, since $\min(l_j, r_k) = |just\text{-}match_l(v_i)| \geq 1$, the first L-packet received by v_j (denoted by L_m) and the first R-packet received by v_k (denoted by R_m) will have the same identifier list, i.e., $\langle L_m \rangle = \langle R_m \rangle$. Moreover, the number of elements in each list is exactly $just\text{-}match_l(v_i)$. By fact 3 again, x_a is also the p^{th} rightmost parenthesis in $unmatch_l(v_j)$ and x_b is the p^{th} leftmost parenthesis in $unmatch_r(v_k)$. Then using lemma 7, we can conclude that x_a and x_b are assigned with the same identifier. □

4 Time and Processor Complexity

The following lemma shows that the completion time of a node is no more than 5 time units after the completion of its parents.

Lemma 9 If node v_i receives its last packet at time t_i, then its children will receive all their packets no later than time $t_i + 5$.

Proof: Without loss of generality, let us refer to figure 3 and assume that $l_j \leq r_k$. Since only one packet can be received or sent out in one time unit and t_i is the time that node v_i receives its last packet, M_i must have been received by time $t' = t_i - (x + y)$. Then according to our algorithm, M_j and M_k will be sent to nodes v_j and v_k at time no later than $t' + 1$ and $t' + 2$, respectively; and R_m to node v_k at $t' + 3$ at the latest. Similarly, $R_1, \cdots, R_{w-1}$ should be received by $t'+1, \cdots, t'+w-1$ and can be relayed to node v_j no later than $t'+4, \cdots, t'+w+2$. With similar arguments, packets $R_{w'}, L_m$ (to node v_j) and $R_{w''}$ (to node v_k) can be sent out by time $t' + w + 3, \cdots, t' + w + 5$, followed by $R_{w+1}, \cdots, R_x, L_1, \cdots, L_y$ by time $t' + w + 6, ..., t' + x + y + 5$ at the latest. Thus the lemma is proved. □

Theorem 10 The whole algorithm requires at most $O(\log n)$ time and $n/\log n$ processors.

Proof: In the initial stage of the algorithm, since each segment have $2 \log n$ consecutive parentheses, it takes at most $O(\log n)$ time for a leaf processor to handle the locally matched parentheses and compute the triple for the attached segment. In the upward stage, the triple of an internal node can be computed in constant time when the triples of its children are available. As the depth of the processor tree is no more than $\log n + 1$, this stage can be completed in $O(\log n)$ time.

The processing of the downward stage consists of two phases. In the first phase, packets flow from the root to the leaves. In the second phase, the indices of parentheses are stored in arrays R and L. Then the array M is computed. For the first phase, lemma 9 implies that a node completes in at most 5 time units after the completion of its parent. Since there are at most $\log n$ levels in the tree, this phase can be completed in $5 \log n$ time units. With $n/\log n$ processors, it is easy to see that the second phase can also be completed in $O(\log n)$ time. □

5 Conclusion

We have shown in this paper that parenthesis matching can be solved in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM, which is an improvement over the previously known methods [BaVi, DeSa] which either require a CREW PRAM

or have a higher-order time or processor complexity. Since the steps other than the matching of parentheses involved in the tree construction [BaVi] require at most $O(\log n)$ time and $n/\log n$ processors on EREW PRAM, with our result on parenthesis matching, the whole process in the computation tree form construction can be performed in the same parallel time and processor complexity on EREW PRAM.

In the upward and downward stages of our algorithm, the processors are organized in the form of a binary tree and information is only passed along the tree edges. It is because of this processor organization, our algorithm can be run on an EREW PRAM without read conflicts and in fact can also be implemented on the tree machine suggested by Bentley and Kung [BeKu] if the manipulation of the arrays R, L and M, is handled separately. It is not sure whether parenthesis matching can be done with a more restricted model than PRAM.

Acknowledgements

The authors thank Andrew Choi for his careful reading of the first draft of this paper.

References

[BaVi] I. Bar-On and U. Vishkin, Optimal Parallel Generation of a Computation Tree Form, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, pp. 384-357.

[BeKu] J.L. Bentley and H.T. Kung, A Tree Machine for Searching Problem, *IEEE 1979 International Conference on Parallel Processing*, August 1979, pp. 257-266.

[DeSa] E. Dekel and S. Sahni, Parallel Generation of Postfix and Tree Forms, *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 300-317.

[Eck] D.M. Eckstein, Simultaneous Memory Access, TR-79-6, Computer Science Dept., Iowa State University, Ames, 1979.

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

[Gold] L.M. Goldschlager, A Unified Approach to models of synchronous Parallel machines, *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 1978, pp. 89-94.

[Knu] D.E. Knuth, A History of Writing Compilers, *Comput. Autom.*, 1982, pp. 6-19.

[Reif] J. Reif, Parallel Time $O(\log n)$ Acceptance of Deterministic CFLs, *Proceedings 23rd IEEE Symposium on Foundations of Computer Science*, (1982), pp. 290-296.

[Vis] U. Vishkin, Implementation of Simultaneous Memory Address Access in Models that Forbid It, *J. Algorithm* Vol. 4, No. 1 (1983), pp. 45-50.