Efficient EREW PRAM Algorithms for Parentheses-Matching

Sushil K. Prasad, Member, IEEE, Sajal K. Das, Member, IEEE, and Calvin C.-Y. Chen

Abstract—We present four polylog-time parallel algorithms for matching parentheses on an exclusive-read exclusive-write (EREW) parallel random-access machine (PRAM) model. These algorithms provide new insights into the parentheses-matching problem. The first algorithm has a time complexity of \(O(\log^2 n)\) employing \(O(\frac{n}{\log n})\) processors for an input string containing \(n\) parentheses. Although this algorithm is not cost-optimal, it is extremely simple to implement. The remaining three algorithms, which are based on a different approach, achieve \(O(\log n)\) time complexity in each case, and represent successive improvements. The second algorithm requires \(O(n)\) processors and working space, and it is comparable to the first algorithm in its ease of implementation. The third algorithm uses \(O\left(\frac{n}{\log n}\right)\) processors and \(O(n\log n)\) space. Thus, it is cost-optimal, but uses extra space compared to the standard stack-based sequential algorithm. The last algorithm reduces the space complexity to \(O(n)\) while maintaining the same processor and time complexities. Compared to other existing time-optimal algorithms for the parentheses-matching problem that either employ extensive pipelining or use linked lists and comparable data structures, and employ sorting or a linked list ranking algorithm as subroutines, our last two algorithms have two distinct advantages. First, these algorithms employ arrays as their basic data structures, and second, they do not use any pipelining, sorting, or linked list ranking algorithms.

Index Terms—EREW PRAM, optimal algorithms, parallel algorithm, parentheses-matching, parsing

I. INTRODUCTION

The parentheses-matching problem is used to determine the mate of each parenthesis in a balanced string of parentheses. A string of \(n\) parentheses can be matched sequentially in \(O(n)\) time using a stack by scanning the input string once. Since parentheses-matching is an integral subproblem of parsing and evaluating expressions by computation-tree generation [3], [4], [16], [20], several parallel algorithms have been proposed on the parallel random-access machine (PRAM) model [7], [4], [5], [7], [8], [14], [18], [19], [24], [27], [31], [33], and some on distributed memory models, such as mesh, shuffle exchange [32], and hypercube networks [9]. More interestingly, none of these algorithms makes explicit use of a stack. For additional literature related to this problem, readers may refer to an annotated bibliography on parallel parsing [1] and an in-depth survey of parallel parentheses-matching algorithms [28].

The PRAM model consists of several processors and a global shared memory, in which each processor can access a shared memory cell in constant time. Arithmetic and logical operations can also be performed in unit time. There is a global clock. All synchronizations and communications take place via the shared memory. Among different variants of the PRAM family, we consider the weakest, albeit the most feasible, model, namely, the exclusive-read and exclusive-write (EREW) variant, in which no two processors can simultaneously read from or write into a memory cell. In the concurrent-read and exclusive-write (CREW) model, however, simultaneous reading, but not simultaneous writing, from a memory cell by more than one processor is allowed. The concurrent read and concurrent-write (CRCW) model allows simultaneous reading and simultaneous writing.

Although the sequential algorithm for parentheses matching is straightforward and optimal, designing a parallel algorithm is nontrivial, which has given rise to a number of fast algorithms for CREW and EREW PRAM's, several of which are cost-optimal. In this paper, we present four EREW algorithms for parentheses-matching that provide new insights into this interesting problem and that are easier to implement compared to the competing algorithms. They are presented in the order of successive improvements in their resource requirements (i.e., time, processor, and working space).

Algorithm I is based on a partitioning strategy, exploiting a simple characterization of a balanced string of parentheses. Compared to the existing parallel algorithms for parentheses-matching, Algorithm I is the simplest, though costwise it is nonoptimal. This algorithm segregates parentheses by nesting levels, placing odd-level parentheses to the left of even-level parentheses, and packing the result. Each such partitioning requires \(O(\log n)\) time, and \(\log n - 1\) iterations are sufficient to bring the matching pair of parentheses adjacent to each other. Thus, the time required is \(O(\log^2 n)\), and the space used is \(O(n)\). A preliminary version of this result is available in [14]. Algorithm I is the content of Section II.

Section III describes Algorithm II as our first example of a match/copy EREW algorithm, implementing the central idea in its simplest form. Although not cost-optimal, this algorithm requires \(O(\log n)\) time employing \(O(n)\) processors, and it serves as a helpful introduction to the more involved cost-optimal EREW algorithms described later.
Algorithm III is a cost-optimal EREW algorithm, which builds a "copying/matching" tree whose nodes represent the unmatched parentheses in their spanned substrings using an encoding scheme. At each pair of sibling nodes, matching is performed, and the remaining unmatched parentheses are copied into the parent node. This is done in \(O(1)\) time by means of a suitable array data structure, yielding an overall time of \(O(\log n)\). However, the total working space is \(O(n \log n)\). Preliminary versions of this algorithm previously appeared in [17], [18]. This algorithm is detailed here in Section IV.

Section V describes Algorithm IV, which is obtained by combining previous methods. This is a relatively simple, cost-optimal EREW algorithm with complexities of \(O(\log n)\) time and \(O(n)\) space using \(O(\frac{n}{\log n})\) processors. A "virtual" copying/matching tree is produced that is encoded as in Algorithm III, but is stored in a distributed fashion among the various processors. By using the positional matching trick of Tsang, Lam, and Chin [33], and by ensuring "EREWness" by redundant positional address calculation via a parallel prefix-sum algorithm, we avoid actually building the matching tree of Algorithm III, thus achieving greater simplicity in less space. A preliminary version of this result appeared in [8].

Section VI briefly surveys the existing time-optimal EREW algorithms for the parentheses-matching problem and compares them with our time-optimal algorithms. Our algorithms use only arrays for storage. Furthermore, our algorithms do not employ pipelining, or involved subalgorithms with large constant of proportionality, such as linked list ranking [11], [12], or parallel sorting [10]. Thus, our solutions also serve to demonstrate that the parentheses-matching problem is easier than both sorting and linked list ranking problems on the EREW PRAM model.

Section VII contains some concluding remarks.

Before proceeding further, we outline a few conventions. All logarithms are assumed to be in base 2. We will use C language conventions. Let \(n\) input parentheses be available in an array \(\text{PAREN}[1, \ldots, n]\) at the beginning of each algorithm, where \(n\) is even. Output is to be obtained in an array \(\text{MATCH}[1, \ldots, n]\) such that \(\text{MATCH}[i] = j\) implies that the parentheses \(\text{PAREN}[i]\) and \(\text{PAREN}[j]\) are mates, for \(1 \leq i \neq j \leq n\).

The well-known prefix-sum problem will occur as a sub-problem on several occasions. Given an array of \(n\) elements \(x_1, x_2, \ldots, x_n\), this problem computes all the partial sums \(\sum_{i=1}^{k} x_i\), for \(1 \leq k \leq n\). It can be optimally solved in \(O(\log n)\) time employing \(O(\frac{n}{\log n})\) processors on the EREW PRAM model [11], [26].

II. Algorithm I: An Odd Even Segregation Algorithm

Consider the following two observations concerning a balanced input string.

1) The mate of a parenthesis at an odd position in a balanced input string lies at an even position.

2) If a balanced string does not have any left parenthesis at an even position (or, equivalently, a right parenthesis at an odd position), then the mate of each left parenthesis in the string lies immediately to its right.

One can show that any string satisfying the second observation is of the following form:

\[
(\ldots(\ldots(\ldots)\ldots))
\]

Algorithm I is formulated by using a partitioning approach as follows. By the first observation, the set of left parentheses at odd positions have their mates in the set of right parentheses at even positions. These parentheses can be matched independently of the matching of the set of left parentheses at even positions and the set of right parentheses at odd positions. Thus, the input string can be partitioned into two substrings, with the first substring containing the left and the right parentheses at, respectively, odd and even positions, and with the second substring containing the left and the right parentheses at, respectively, even and odd positions. If this partitioning fails to split the input string (i.e., if the right substring is empty), then, by the second observation, a balanced input string will have the form \(F\). In this case, all the mates can easily be found in parallel. Otherwise, the partitioning scheme can be repeatedly applied until form \(F\) is obtained.

Fig. 1 illustrates Algorithm I. In this figure, symbols \((i\) and \(j)\) denote a left and right parenthesis at the \(i\)th and \(j\)th positions, respectively. Algorithm I is stated as follows.

1) for \(i = 1\) to \(\lceil \log n \rceil - 1\) do

   a) Mark each left parenthesis at an odd position and each right parenthesis at an even position.
b) Use parallel prefix algorithm to pack the marked parentheses followed by the unmarked parentheses.

2) Check if the input string has been converted to form \( F \). If not, then the input string is unbalanced. Otherwise, match the parentheses, and store the results in the output array MATCH.

Since each substep of Algorithm I can be implemented in \( O(\log n) \) time using \( \frac{n}{\log n} \) processors on an EREW PRAM, and since there are \( \lceil \log n \rceil - 1 \) iterations, Algorithm I requires \( O(\log^2 n) \) time employing \( \frac{n}{\log n} \) processors.

We informally show that \( \lceil \log n \rceil - 1 \) iterations are sufficient for Algorithm I. For a rigorous proof of this claim, see [15]. Consider the graphical representation of an input string, as shown in Fig. 2. The parentheses belonging to a level can be matched independently of any other level. The level of a left parenthesis at a position \( i \) is given by the difference in the number of left and right parentheses in the substring \( \text{PAREN}[1 \ldots i] \). The level of a right parenthesis at position \( j \) is one more than the difference in the number of left and right parentheses in \( \text{PAREN}[1 \ldots j] \). The partitioning scheme places the parentheses belonging to the odd levels of each substring in the left partition, and those belonging to the even levels in the right partition. This phenomenon is illustrated in Fig. 2. Thus, if the input string has \( l \) levels, its left partition would have \( \lceil \frac{l}{2} \rceil \) levels, whereas its right partition would have \( \lfloor \frac{l}{2} \rfloor \) levels. Furthermore, since the maximum number of levels in a balanced string of \( n = 2^k \) parentheses is \( 2^{k-1} \) (in a string of \( 2^{k-1} \) left parentheses followed by \( 2^{k-1} \) right parentheses), the number of levels in the left and right partitions of a string cannot exceed \( 2^{k-2} \). Subsequent \( (k - 2) \) repeated partitionings of the substrings would reduce the number of levels in any substring to \( 2^0 = 1 \), thus converting the input string to the desired form \( F \).

It can also be shown that the average number of partitioning steps is \( \Theta(\log n) \). Interested readers may obtain a proof by using a result on the average height of planted plane trees [6], [15].

Remarks: Can we make Algorithm I any faster? The answer lies in the observation that Algorithm I converts an input string eventually to form \( F \) by partitioning the input string into its constituent Level-I substrings, one for each level. Instead of repeated partitioning to obtain form \( F \), one can determine the level of each parenthesis by using a parallel prefix algorithm, and then stable sort the parentheses according to their levels. Applying Cole's parallel merge sort algorithm [10], this can be accomplished in \( O(\log n) \) time using \( O(n) \) processors on an EREW PRAM model.

A reduction in the number of processors would be possible if \( \frac{n}{2} \) integers, each representable in \( O(\log n) \) binary bits, can be (stable) sorted in \( O(\log n) \) time using fewer than \( \Theta(n) \) processors. Using a parallel bucket sort algorithm [11], [12], [34], the time complexity of such a parentheses-matching algorithm would be \( O(\frac{\log n}{\log \log n}) \) employing \( O(\frac{n}{\log n}) \) processors. To our knowledge, however, there is no algorithm that sorts \( n \) integers in \( [1, \cdots, n] \), each representable in \( O(\log n) \) bits, in \( O(\log n) \) time using \( O(\frac{n}{\log n}) \) processors on an EREW PRAM.\(^1\) Therefore, this approach falls short of cost-optimality. But the close relationship between matching parentheses and sorting integers has lead us to the following result. Using an optimal parentheses-matching algorithm (such as ones presented later in this paper) as a subroutine, we have developed an \( O(\log n) \)-time \( O(\frac{n}{\log n}) \)-processor algorithm on the EREW PRAM to sort a special class of integers in which any two successive integers in the input sequence differ at most by 1 [7], [9]. A similar result has been independently reported by Diks and Rytter [19].

III. ALGORITHM II: AN EFFICIENT MATCH-AND-COPY ALGORITHM

We now discuss an alternate approach to parentheses-matching problem that turns out to be more fruitful. In a substring of parentheses, when all the locally matched pairs of parentheses are found and removed, the substring is reduced to a standard form consisting of the unmatched right parentheses followed by the unmatched left parentheses [20]. If a substring is balanced, its reduction will leave it empty. Consider the

\(^1\) A result [21] sorts \( n \) integers drawn from \([1, \cdots, n]\) in \( O(\log n) \) time employing \( O(\frac{n}{\log n}) \) processors, only if the basic operations on \( O(n \log n) \)-bit integers are assumed to require unit time. Likewise, an algorithm by Rajasekaran and Sen [30] requires a word length of \( n^\epsilon \), for any \( \epsilon > 0 \).
following recursive algorithm to reduce, and thus match, an input string.

1) Split the string into two halves. Recursively reduce the left and the right substring.
2) Match the reduced left and right substrings. If the left substring contains \( r_1 \) right parentheses followed by \( l_1 \) left parentheses, and if the right substring contains \( r_2 \) right parentheses followed by \( l_2 \) left parentheses, match the rightmost \( z = \min(l_1, r_2) \) left parentheses of the left substring with the leftmost \( z \) right parentheses of the right substring. Remove those \( z \) matched pairs, leaving \( r = r_1 + r_2 - z \) right parentheses followed by \( l = l_1 + l_2 - z \) left parentheses.

A. Implementation

Suppose the reduced left and right substrings after the first step are formatted in the input array \( PAREN[1, \ldots, n] \) as shown in Fig. 3. Further let there be \( n \) processors, \( P_1 \) through \( P_n \). Then, in Step 1, \( P_1 \) through \( P_{n/2} \) recursively reduce the left substring, and at the end of Step 1, each processor knows the values \( r_1 \) and \( l_1 \). Similarly, processors \( P_{n/2+1} \) through \( P_n \) reduce the right substring and each knows the values \( r_2 \) and \( l_2 \). Step 2 is easily implemented in-place in \( O(1) \) time as follows.

1) For \( i = 1, 2, \ldots, n/2 \), processor \( P_i \) sends its \( r_1 \) and \( l_1 \) values to \( P_{i+n/2} \) and receives \( r_2 \) and \( l_2 \) values in return.
2) For \( i = 0, 1, \ldots, \min(l_1, r_2) - 1 \), if \( l_1 \geq r_2 \), then \( P_{n/2-i} \) matches the left parenthesis in \( PAREN[n/2 - i] \) with the right parenthesis in \( PAREN[n/2 + i + 1] \); else, if \( l_1 < r_2 \), processor \( P_{n/2+i+1} \) performs this matching. This avoids any read-write conflict while matching.
3) If \( l_1 \geq r_2 \), then for \( i = r_2 \) through \( l_1 \), processor \( P_{n/2-i} \) moves \( PAREN[n/2 - i] \) to \( PAREN[n-i-2 + r_2 - i] \). Else, if \( l_1 < r_2 \), then for \( i = l_1 \) through \( r_2 \), processor \( P_{n/2+i+1} \) moves \( PAREN[n/2 + i + 1] \) to \( PAREN[r_1 + l_1 - i + 1] \).

Since the left and the right substrings in Step 1 are reduced recursively, the total time required, \( T(n) \), is given by the recurrence relation \( T(n) = T(n/2) + O(1) \). Thus, Algorithm II is an in-place, \( O(\log n) \)-time algorithm using \( O(n) \) processors. Fig. 4 shows the working of Algorithm II as an iterative algorithm (after unfolding the recursion to substrings of size 1) with an example string of 32 parentheses. We use the notation \( (i,j) \) to denote that the left parenthesis \( i \) is matched with the right parenthesis \( j \). The algorithm conceptually builds a binary matching tree in which two siblings containing reduced substrings are matched to obtain the reduced substring of their parent.

Remarks: One can clearly state Algorithm II iteratively, wherein processor \( P_i \) starts with the parenthesis \( PAREN[i] \), and after each matching iteration, the size of a substring doubles. The details can be filled in easily. Similar to Algorithm I, this algorithm is conceptually straightforward. After modifications, Algorithm II leads to a cost-optimal algorithm, which is described next.

IV. ALGORITHM III: AN OPTIMAL MATCH-AND-COPY ALGORITHM

In Algorithm II, all the unmatched parentheses are carried from level to level in the binary matching tree. Therefore, in the worst case, the work performed at each level is \( O(n) \). In Algorithm III, however, we reduce this work from \( O(n) \) to \( O(\log n) \). Correspondingly, we use only \( \frac{n}{\log n} \) processors instead of \( n \), yet keeping the time spent at each level at \( O(1) \).

We briefly indicate the key ideas for Algorithm III in this paragraph, and give pertinent details later. In the last level of the matching tree of Fig. 4, the left sibling has six unmatched left parentheses, \( (1, 2, 3, 6, 15, 16) \), which can each be matched by six processors in \( O(1) \) time with the matching right parentheses of the right sibling, which contains \( (17, 26, 27, 30, 31, 32) \).
Suppose the left parentheses, 1, 2, 3, are collectively represented by the notation (13, which denotes these three left parentheses with 1 as their leftmost. In general, let a representative parenthesis (j denote j left parentheses, with 1 being the leftmost, and, similarly, let j ) denote j right parentheses, with j being the rightmost.2 Following such a scheme, the left sibling may be represented by (13 (6 1 (152, and the right sibling may be represented by 3)a7 2)a7. Such encoded siblings can be sequentially matched by a processor by starting with the leftmost left representative parenthesis (13 and the rightmost right representative parenthesis 3)a7, and matching (1 with )a7, then (2 with )a7, and (3 with )a7. Then, using the representative parentheses (61 and 2)a7, matching (6 with )a7, then, using the left representative parenthesis (152 while using 2)a7, matching (15 with )a7, and finally, using the right representative parenthesis 1)17, matching (17 with )17. As illustrated by this matching procedure, a representative parenthesis can be used to index the parentheses it represents. Next the left sibling can be partitioned into two segments, (13 and (61 (152, each representing three parentheses, and likewise, the right sibling can be partitioned into 3)a7 and (1)27 1)a7. Therefore, two processors can be employed to concurrently match the corresponding left and right segments: (13 with 3)a7, and (61 (1)27 1)a7. Similarly, the left sibling can be partitioned into three segments representing two parentheses each: (12, (31 (61, and (152. Notice that in doing so, the representative parentheses (13 had to be split into (62 and (1. Similarly, the right sibling is partitionable into segments 1)27 1)27 1)30, and (6)23. Here 2)27 was split into 1)27 1)27, and 3)32 was split into (1)27 1)27. Now three processors can independently match (12 with 2)27, (3 1)61 with 1)27 1)30, and (152 with 1)17 1)17. We note two points here.

1) Multiple processors are employable to match a pair of encoded siblings if the siblings can be kept partitioned appropriately.

2) Since matching a pair of left and right partitions starts with the leftmost, representative parenthesis of the left partition and the rightmost representative parenthesis of the right partition, these two representative parentheses in particular are essential for their respective partitions. Furthermore, to create the leftmost representative parenthesis of a left partition, a representative parenthesis of the preceding partition might be required to be split. Similarly, a representative parenthesis of a right partition might be required to be split to create the rightmost representative parenthesis of the succeeding right partition.

Algorithm III employs an encoding/indexing scheme to avoid carrying all the unmatched parentheses from level to level in the binary matching tree, thus reducing the work from \(O(n)\) to \(O(\log n)\) per level. It also keeps the encoded nodes partitioned into \(\log n\)-size segments so that after the entire matching tree has been constructed, and after all the matching segment-
pairs have been identified, each pair of matching left and right segments can be matched independently in $O(\log n)$ time.

Assuming an input string of length $n = 2^m$, Algorithm III uses $p = 2^q$ processors, where $q = \lceil \log[n/m] \rceil$. This algorithm consists of three phases, each requiring $O(\log n)$ time. In Phase 1, the input string is partitioned into $p$ substrings, each of size $k = 2^{m-q}$. Each processor reduces its substring sequentially; i.e., it finds the locally matched parentheses, and the resultant reduced substrings are made available in the input array PAREN, as shown in Fig. 5. In Phase 2, all $2^q$ substrings are matched iteratively to construct a binary matching tree of height $q$. In Phase 3, each pair of matching left and right segments is matched.

A. Data Structure and Operations

The matching tree is stored in an array TREE[$1, \ldots, q + 1|[1, \ldots, n]$, where TREE[$i$] stores the $i$th level. TREE[$1$] stores the leaves such that the $i$th leaf occupies the subarray TREE[$1$][(i - 1)k + 1, \ldots, ik], for $1 \leq i \leq q$. Thus, each leaf has a capacity $k$ to accommodate up to $k$-size subsequences produced from Phase 1. Each parent at a level $i + 1$ is double the size of its children at level $i$, occupying those positions of TREE[$i + 1$] that both its children occupy in TREE[$i$]. Each node of the tree is conceptually kept partitioned into $k$-size segments.

At the beginning of Phase 2, processor $P_i$ is assigned to the $i$th reduced substring produced by Phase 1, for $1 \leq i \leq p$. A processor encodes its substring into two representative records: the right representative record, $rrep$, contains the index of the rightmost right parenthesis of the substring in the array PAREN ($rrep$.index), and also the total number of right parentheses of the substring ($rrep$.size). Similarly, the index of the leftmost left parenthesis and the total number of left parentheses form the left representative record, $lrep$. A representative record contains information for the conceptual representative parentheses described earlier.

Each processor $P_i$ is responsible for carrying its representative parentheses along the matching tree. $P_i$ keeps track of the current location of its $rrep$ by a local variable $lpos$ and that of its $rrep$ by $rpos$. Depending on the location of its representative records, a processor is associated with either a left or a right sibling. Processor $P_i$ also keeps, for a left sibling node, a local copy of the total number of unmatched right parentheses in $r_1$, and that of the left parentheses in $l_1$. Likewise, a processor $P_i$ associated with a right sibling keeps the corresponding values in local variables $r_2$ and $l_2$. 
Each representative record occupies the same position in each node that the corresponding parenthesis would occupy if Algorithm II were used. Therefore, the calculation to place the representative records (i.e., updating lpos and rpos) from a child to its parent is similar. Analogously to Algorithm II, as one moves down the matching tree, fewer unmatched parentheses are carried along leaving those parentheses that matched, correspondingly, the size of the representative records are reduced to represent increasingly fewer unmatched parentheses.

A record lrep of a processor $P_j$ placed at a position $\text{TREE}[i][\text{lpos}]$ represents the left parentheses corresponding to locations lpos through (lpos+lrep.size -1) of $\text{TREE}[i]$. If this range of locations goes beyond the boundary of the segment containing location lpos, processor $P_j$ fills in the leftmost location of the succeeding left segment with an appropriate temporary record. As indicated earlier, the leftmost position of each left segment is to be filled with a record, because it is required to initiate the matching of that segment. Similarly, if rrep’s range crosses the boundary of its segment, the associated processor stores a temporary record at the rightmost position of the preceding right segment.

Finally, if a processor finds that its representative record is matched, it discontinues carrying that record further down the matching tree. Furthermore, if a matched left (right) representative record is the rightmost (leftmost) representative record in its segment, the associated processor becomes responsible for matching that segment with the symmetrically opposite segment of the sibling. To avoid access conflict between the processors of left and right segments during Phase 3, matching is performed by the processor associated with the right sibling if $r_2 < l_1$; else, the matching is performed by the processor associated with the left sibling. Thus, after Phase 2 is over, a processor is assigned to match at most two pairs of left and right segments, one pair for each of its representative records.

Fig. 5 shows the entire matching tree constructed during Phase 2. Above each representative record, the processor responsible for copying it is also shown. The shaded regions of each pair of siblings have been found to be matching, and the processor that would match specific segment pairs during Phase 3 are shown encircled.

**B. Detailed Steps of Algorithm III**

**Phase 1:** For $i = 1, 2, \ldots, p$, processor $P_i$ sequentially finds the locally matched parentheses in subarray PAREN[(i-1) * $k + 1, \ldots, (i * k)]$. The reduced substrings are left in its subarray by packing the right beginnings at the parent and the left parentheses at the end of the subarray. $P_i$ constructs its representative records rrep and lrep, initializes its local variables $r_1$ and $l_1$ if $i$ is odd (i.e., $P_i$ is at a left sibling); otherwise, it initializes $r_2$ and $l_2$. Processor $P_i$ sets the positions of its records rpos as the index of the rightmost right parenthesis in the array PAREN, and, similarly, lpos as the index of the leftmost left parenthesis. Additionally, $P_i$ keeps two private variables:

1) the number, rtu, of right parentheses represented by its rrep and by those to its left in a node; and, likewise 2) the total number, ltu, of the left parentheses represented by its lrep and by those to its right. Initially, $r_u = rrep.size$ and $l_u = lrep.size$.

**Phase 2:**

1) **Initialize the leaf nodes in TREE[1]:**

- $P_i$ stores its representative records, rrep, at TREE[1] [rpos], and lrep at TREE[1] [lpos].

2) **Construct the binary matching tree:***

   **For level = 1 to q do**

   [match the nodes at TREE[level] and construct TREE[level+1]]

   a) **Exchange r’s and l’s between the processors of siblings:** There are $t = 2^{\text{level}-1}$ processors.

   - $P_{(j-1)+1}$ through $P_{i+j}$ belonging to node $j$ at this level of the tree. For $j = 1, 2, \ldots, p$. Node $j$ is a left sibling if $j$ is odd. Processor $P_i$ of a left sibling sends its $r_1$ and $l_1$ to $P_{i+j}$, of the right sibling, and receives $r_2$ and $l_2$ in return.

   b) **Copy the unmatched representative records in the parent:**

   - $P_i$, belonging to a left sibling copies its rrep in exactly the same location in TREE[level + 1] as it was in TREE[level]. $P_i$, of a right sibling copies its lrep similarly.

   - $P_i$ of a left sibling, if $rtu > r_u$, computes new

   - $lrep.size = \min(lrep.size, ltu - r_u)$, $ltu = ltu - r_u + l_2$, and copies lrep to its new position $lpos = lpos + l_2 - k2^{\text{level}-1}$ in TREE[level +1].

   Analogously, for the rrep of a processor belonging to a right sibling, if $ltu < r_u$, $rrep.size = \min(rrep.size, ru - l_1)$, $ru = ru - l_1 + r_1$, and $rpos = rpos - l_1 - k2^{\text{level}-1}$.

   c) **Fill in the temporary records, if necessary:**

   - The relative position of rrep of a processor in its segment (from left) at the parent node is

   - $z = ((ru - 1) \mod k) + 1$. If $z < rrep.size$, a temporary record with size $= rrep.size - z$ and index $= rrep.index - z$ is filled in the last position $\text{TREE}[\text{level} + 1][\text{rpos} - z]$ of the preceding segment. Similarly, the relative position of its lrep in its segment from right is

   - $z = ((ltu - 1) \mod k) + 1$. If $z < lrep.size$, a temporary record with size $= lrep.size - z$ and index $= lrep.index + z$ is stored at $\text{TREE}[\text{level} + 1][\text{lpos} + z]$.

   d) **Assign processors to matching segments:**

   - If $r_2 < l_1$ {processors assigned to the right sibling will perform the matching task}, then, for $P_i$ of a right sibling, if $r_u < r_2 \{rrep has a match in the left sibling\}$, then: if $((ru - 1) \mod k) + 1 \leq rrep.size \{rrep is the leftmost record of this segment\}$. $P_i$, during Phase 3, has to match the right segment containing rrep, with the matching left segment being in the left sibling. Accordingly, $P_i$ saves a copy of all its current variables, and the variable level. {Analogous processing for a left sibling}.
Fig. 6. (a) Number of parentheses matched at each node. (b) Calculation of INDEX[1 level][l p].

Phase 3: For i = 1, 2, . . . , p, if P_i has been assigned to match a pair of segments, P_i sequentially matches it. In Step 2d of Phase 2, let processor P_i of a right sibling be assigned to match a right segment with the symmetrically located left segment of the left sibling.

1) Find the position of the rightmost and leftmost records, respectively, in the right and the left segments:
   r = \min \{ \text{last position} \lfloor \frac{\text{rpos}}{k} \rfloor k \text{ of the right segment} \}
   and
   l = \text{max} \{ \text{first position} \lceil \frac{\text{lpos}}{k} \rceil \lfloor \frac{\text{lpos}}{k} \rfloor t + 1 \text{ of the left segment} \}
   where t = 2^{k \text{level}-1} \text{ is the size of a node at this level.}

2) Find the left and right representative records at symmetrically opposite locations:
   The position symmetric to r of right segment in the left segment is l' = 2 \lfloor \frac{\text{rpos}}{k} \rfloor t - r + 1. Assuming that l < l',
   \{ analogously, update rep if l > l' \}
   While l < l', \text{ rep} = \text{TREE}[\text{level}][l]; l = l + \text{rep.size} \{ endwhile \}

If l > l', then modify current rep as
   \text{rep.size} = \text{rep.size} - (l - l'); \text{ rep.index} = \text{rep.index} + (l - l'); l = l'.

3) Match the segments starting with \text{rep} = \text{TREE}[\text{level}][l] and \text{rep} = \text{TREE}[\text{level}][r]:

While l < r do
   a) Match the right parenthesis in \text{PAREN}[\text{rep.index}]
      with the left parenthesis in \text{PAREN}[\text{rep.index}].
   b) Decrement \text{rep.index} and increment \text{rep.index}.
      Decrement \text{rep.size}, \text{rep.size}, and r, and increment l.
   c) If \text{rep.size} = 0, read the new \text{rep} from \text{TREE}[\text{level}][r], similarly, if \text{rep.size} = 0, read new \text{rep} from \text{TREE}[\text{level}][l].

Remarks: Notice that the private variables \text{lu} and \text{ru} can be derived as \text{ru} = (\text{rpos} - 1) \mod (k2^{k \text{level}-1}) + 1 and \text{lu} = k2^{k \text{level}-1} - (\text{lpos} - 1) \mod (k2^{k \text{level}-1}) + 1 from \text{rpos} and \text{lpos}, respectively. These two variables have been used only to clarify the presentation. It is easy to conclude that each of the Phases 1, 2, and 3 requires \(O(\log n)\) time using \(n \log n\) processors. Therefore, Algorithm III is a cost-optimal EREW PRAM algorithm. The space usage, in contrast with Algorithm II, is \(O(n \log n)\), because each level of TREE requires \(O(n)\) space. The space complexity, therefore, is not optimal compared to \(O(n)\) stack-space used for a sequential algorithm. The next algorithm cleverly modifies the implementation aspects of Algorithm III to bring down the space complexity back to \(O(n)\) while maintaining the parallel time at \(O(\log n)\), employing \(n/\log n\) processors.

V. ALGORITHM IV: A TIME- AND SPACE-OPTIMAL ALGORITHM

The key idea of Algorithm IV is not to construct the binary matching tree explicitly, because there are many empty spaces in the matching tree that remain unused (Fig. 5). The portions of the matching tree that are of use during Phase 3 are the matched portions of different nodes. The unmatched portions serve only to calculate the parent nodes during Phase 2. Algorithm IV, therefore, uses two arrays, \text{LMATCH}[1 \ldots k/2] and \text{RMATCH}[1 \ldots k/2], to store the matching portions of the left and right siblings, respectively. The sizes of the arrays \text{LMATCH} and \text{RMATCH} are precisely \(k/2\) each, so both can accommodate all the left and right parentheses, respectively. In these arrays, the matching portions of left and right siblings are stored aligned in such a way that the left parenthesis at \text{LMATCH}[l] is the match of the right parenthesis at \text{RMATCH}[r], for \(1 \leq l \leq r \leq k/2\). Thus, during Phase 2 of Algorithm IV, although each processor carries its two representative records of the "virtual" matching tree level-by-level, it does so only in its private memory while keeping track of the current level of the matching tree and the particular node at that level. Whenever the whole or a part of a representative record is found to be matched in the corresponding sibling, the representative record (or a temporary record, if only a part has matched) is stored in \text{LMATCH} (RMATCH) if the current node is a left (right) sibling at an appropriate location.
Calculation of the appropriate location will be explained in later paragraphs.) During Phase 3, equal portions of the arrays LMATCH and RMATCH are allocated to all the processors, and each processor sequentially matches the parentheses in its subarrays.

As in Algorithm III, assuming an input string of length \( n = 2^m \), Algorithm IV uses \( p = 2^q \) processors, where \( q = \lceil \log(n/m) \rceil \), and it also consists of three \( O(\log n) \)-time phases. Phase 1 is same as in the previous algorithm. For Phase 2, let us number the nodes of the matching tree starting from levels containing the leaves down to the level of the root, and within each level from left to right. Therefore, the leftmost leaf is node 1, and the rightmost leaf is node \( p \) at level 1; the leftmost node at level 2 is node \( p + 1 \); and the rightmost is node \( p + 2 \), and, likewise, the root is node \( 2p - 1 \). Thus, an odd-numbered node (except the root) is a left sibling. The primary additional task here, in addition to what Algorithm III does during its Phase 2, is to calculate the proper indices into arrays LMATCH and RMATCH where the matched representative records are to be stored for each pair of siblings of the matching tree. A processor associated with node \( i \) of the matching tree would need to know the exact number of parentheses matched at nodes 1 through \( i \) in order to store the parentheses matched at node \( i \) into the arrays LMATCH and RMATCH. A 2-D array \( INDEX[1, \cdots, q][1, \cdots, p] \) is used to precalculate and store the above information such that each processor \( P_i \) associated with the \( i \)th node (from left to right) at level \( i \) has a private copy \( INDEX[i][i] \) as the index into arrays LMATCH and RMATCH, for \( 1 \leq i \leq \frac{p}{2^l} \) and \( (j - 1)2^l + 1 \leq i \leq j2^l \), thus avoiding any concurrent read for this purpose.\(^3\)

To calculate and fill in the array \( INDEX \), the number of parentheses matched at each pair of left and right siblings, i.e., the \((2j - 1)\text{th}\) and the \((2j)\text{th}\) node at level \( l \), for \( 1 \leq j \leq \frac{p}{2^l} \) and \( 1 \leq l \leq q \), is calculated and stored in \( INDEX[l][2^l(2j - 1) + 1] \) and \( INDEX[l][2^l(2j) + 1] \). This step does not require actual building of the matching tree, only the values \( l_1, l_2, r_2, \) and \( l_2 \) need to be calculated for each pair of siblings. Other entries of \( INDEX \) are initialized to zeros beforehand. Next a parallel prefix sum algorithm is used on array \( INDEX \) treated as a row-major linear array. The effect of this step is that now the entire subarray \( INDEX[l][2^l(2j - 1) + 1, \cdots, 2^l(2j)] \) contains the required index for the processors associated with the left and right siblings. Fig. 6(b) illustrates this process with the same input string as in Fig. 5. The first element of the tuple

\(^3\) A similar preprocessing is carried out by Anderson et al. [2].
in Fig. 6(a) corresponds to the number of right parentheses, and the second element is the number of left parentheses with each processor.

In the next stage, the matching tree is constructed conceptually level-by-level, and the matching left and right representative records of each pair of siblings are stored into the arrays LMATCH and RMATCH, respectively, by using the indices available in the array INDEX. Those records, which remain unmatched, are updated for the parent node. Fig. 7 illustrates this stage. The vertical dotted lines separate the private variables of each processor. An lrep with fields index, size, and u is shown as (index, size) (and analogously for an rrep). The field u serves as the variable lu of Algorithm III. The records matched are shown enclosed within "(" and ")."

In Phase 3, processor \( P_1 \) is assigned to the subarrays LMATCH\([k(i - 1) + 1, \ldots, k_i] \) and RMATCH\([k(i - 1) + 1, \ldots, k_i] \), for \( 1 \leq i \leq p \). Each \( P_i \) (except for \( P_p \)) reads its rightmost left representative record, lrep, in its portion of the subarray of LMATCH. Let this record be at position \( j \). If \( lrep.size + j - 1 > k_i \), then \( P_i \) writes a new temporary record at LMATCH\([k_i + 1] \). The size of the record lrep is reduced to represent parentheses only through the subarray boundary. Similarly, \( P_i \) fills in a temporary right representative record in RMATCH, if necessary. Now each left and right subarray has its leftmost and rightmost representative record, respectively. Next each \( P_i \) (including \( P_p \)) reads \( lrep = LMATCH\([k(i - 1) + 1] \) and \( rrep = RMATCH\([k(i - 1) + 1] \) and matches the \( k \) left and right parentheses represented by these matching left and right subarrays as in Phase 3 of Algorithm III. Fig. 8 illustrates Phase 3. In this figure, \( P_1 \) and \( P_2 \) fill in temporary records at, respectively, the third and the ninth locations of arrays LMATCH and RMATCH.

A. Detailed Steps of Algorithm IV

Phase 1 is as in Algorithm III. Each processor has variables \( r_1, l_1, r_2, \) and \( l_2 \), and records lrep and rrep. As indicated earlier, each such record has three fields. The fields size and index are as in the previous algorithm, keeping the number of parentheses represented by the record and the index of the leftmost (or the rightmost for rrep) parenthesis in the input array PAREN, respectively. Additionally, a third field lrep.u keeps the total number of left parentheses represented by the record lrep and those represented by the records to the right of lrep in a node. Initially, at a leaf node, lrep is the only record in the node; therefore, \( lrep.u = lrep.size \). Likewise, \( rrep.u \) contains the total number of right parentheses represented by rrep and by those to its left in a node.

Phase 2:

1) Initialize the entries of the array INDEX to zeros.
2) Compute the number of matched parentheses at each pair of siblings, and store into the array INDEX:
   for level = 1 to q do
   for all \( j = 1,3,5,\ldots,p \), where \( t = 2^{level-1} \), processor \( P_t(j-1)+1 \) of a left sibling gets \( r_2 \) and \( l_2 \) values from \( P_{t(j-1)+1} \) of its right sibling in parallel, and stores the number, \( min(l_1, r_2) \), of parentheses matched into INDEX\([level][t(j-1) + 1] \). For the parent, \( P_{t(j-1)+1} \) calculates the new values \( r_1 = r_2 = r_2 - min(l_1, r_2) \)
   and \( l_1 = l_2 = l_1 + l_2 - min(l_1, r_2) \) and \( l_1 = l_2 = l_1 + l_2 - min(l_1, r_2) \).
3) Use parallel prefix-sum algorithm on the array INDEX\([1\ldots q][1\ldots p] \) treated as a row-major linear array.
4) Construct the conceptual matching tree, and store the matching representative records into the arrays LMATCH and RMATCH using indices from array INDEX: (Each processor begins with the same local variables that it had at the end of Phase 1. Arrays LMATCH\([1\ldots \frac{p}{2}] \) and RMATCH\([1\ldots \frac{p}{2}] \) are initialized.)
   for level = 1 to q do
   a) Exchange r’s and l’s among the processors of siblings:
      There are \( t = 2^{level-1} \) processors, \( P_t(j-1)+1 \) through \( P_{t+1} \), belonging to node \( j \) at this level of the tree, for \( j = 1,2,\ldots,p \). Node \( j \) is a left sibling if \( j \) is odd. Processor \( P_i \) of a left sibling sends its \( r_1 \) and \( l_1 \) to \( P_{i+1} \) of the right sibling, and receives \( r_2 \) and \( l_2 \) in return.
   b) Copy the matched records into arrays LMATCH and RMATCH:
      Number of matched parentheses are \( z = min(l_1, r_2) \). For \( P_i \) of a left sibling, if \( lrep.u \leq z \), its lrep is matched completely. Therefore, \( P_i \) copies its lrep into LMATCH\([INDEX][level][i] - lrep.u+1 \]. If \( lrep.u > z \), but \( lrep.u - size < z \), then a part of this lrep has matched. Accordingly, a temporary record is copied into LMATCH\([INDEX][level][i] - z + 1 \) with size = lrep.size - (lrep.u - z), index = lrep.index +...
PRASAD et al.: EFFICIENT EREW PRAM ALGORITHMS

the parentheses represented by subarrays $LMATCH[k(1 - l) + 1, \ldots, ki]$ and $RMATCH[k(i - 1) + 1, \ldots, ki]$, where $k = 2^{m-q}$.

1) Write temporary records in the succeeding subarrays, if necessary:
For $1 \leq i \leq p - 1$, $P_i$ scans its subarrays from left to right and finds the rightmost left representative record $lrep$ in subarray $LMATCH[k(i - 1) + 1, \ldots, ki]$ and the rightmost right representative record in $RMATCH$. If $lrep$ is at location $j$, and if $lrep.size + j - 1 > ki$, then $P_i$ writes a new temporary record at $LMATCH[ki + 1]$ with $size = lrep.size + j - 1 - ki$ and $index = lrep.index + ki - j$. Next, $P_i$ updates $lrep.size = ki - j$. Record $rrep$ is processed similarly.

2) Match the subarrays:
$P_i$ starts with $lrep = LMATCH[k(i - 1) + 1]$ and $rrep = RMATCH[k(i - 1) + 1]$, and matches the $l$ left and right parentheses represented by these subarrays as in Phase 3 of Algorithm III.

Remarks: Algorithms III and IV are both time-optimal, and Algorithm IV is also space-optimal. In this context, we point out that Levcopoulos and Petersson [27] have shown that $O(\log n)$ time is a lower bound for the parentheses-matching problem on the EREW PRAM model.

VI. COMPARISON WITH THE EXISTING COST-OPTIMAL EREW ALGORITHMS

We briefly sketch the existing cost-optimal, $O(\log n)$-time algorithms for parentheses-matching on the EREW PRAM, focusing on the underlying techniques and the data structures employed. We begin, however, with a CREW algorithm by Bar-On and Vishkin [4], because this was the first such PRAM algorithm. It employs $n/\log n$ processors and comprises three $O(\log n)$-time phases. First, each processor is assigned to a log $n$-size substring that is sequentially reduced. The second phase builds a binary tree and then performs concurrent searches on that tree to find mates for the leftmost left and the rightmost right parentheses of each substring. In the third phase, each processor uses the matching found in the second phase to sequentially scan and match the other parentheses in the substring. Each of the leaves of the binary tree of the second phase is assigned to a substring. Each node contains the nesting level of the leftmost left and the rightmost right parentheses in the subsequence spanned by that node. A parent node is easily computed from its children. In the next stage, each processor starts at its leaf and climbs up to a node that “minimally” spans the subsequence containing the mate of the leftmost left parenthesis of its substring, and then climbs down to the appropriate leaf to find the mate. This process is repeated for the rightmost right parenthesis, too. The key data structure used is an array implementing the binary tree.

Now we consider optimal EREW algorithms. To our knowledge, in addition to the proposed Algorithms III and IV, there are four other parentheses-matching algorithms that require $O(\log n)$ time and use $n/\log n$ processors. These are the algorithms due to Anderson, Mayr, and Warmuth [2]; Diks and Rytter [19]; Levcopoulos and Petersson [27]; and Tsang, Lam, and Chin [33]. The first among these is due to Anderson et al. [2]. Having stated this, we must clarify that these four algorithms and our Algorithm III have been developed independently of each other. There also exists an optimal EREW algorithm for a related problem, called the all tallest neighbors problem, due to Kim [24], which can be employed to solve parentheses-matching problem. For completeness, we also discuss Kim’s algorithm.

In Anderson et al.’s algorithm [2], a binary tree of $n/\log n$ nodes is constructed, and each node is assigned to a different processor. Each leaf processor is assigned to a substring of $2\log n$ parentheses, and it begins by reducing its substring sequentially. Then each node calculates a triple $(c(v), m(v), o(v))$, where $m(v)$ is the number of matching pairs found at node $v$, and $c(v)$ and $o(v)$, respectively, are the number of unmatched closing (right) and opening (left) parentheses remaining at node $v$. Separately, assuming an inorder numbering of these nodes, a parallel prefix algorithm is employed to calculate $b(v) = \sum_{w<v} m(w)$ for each node $v$. Next, each matching pair at node $v$ is assigned a unique index $(b(v), i)$, where $i$ is the nesting level of that pair, $0 \leq i \leq m(v) - 1$. Once these indices are communicated to each pair at the leaf level, a left parenthesis is written at the unique location $(b + i)$ of an array, and then it is read by its matching right parenthesis. Communication of the indices requires pipelining. Each node $v$ passes a packet containing $(b(v), 0 \cdots m(v) - 1)$ to its two children. An internal node splits the packet into two by dividing the nesting interval appropriately and passing them down. The leaf processors collect all the packets in their arrival order, and then assign indices to their parentheses. The paper describing this algorithm contains few details of this pipelining process. Among the advantages of this algorithm are its underlying array data structure to implement the binary matching tree and the ease with which this algorithm can be comprehended at an abstract level. However, this algorithm uses extensive pipelining.

The algorithm by Tsang et al. [33] is similar to that of Anderson et al., but the pipelining process used to assign unique identifiers to the matching pairs is too complicated. A possible reason is that Tsang et al. do not precalculate $b(v)$’s.

The Diks and Rytter algorithm [19] has three phases, just as in the Bar-On and Vishkin algorithm [4]. Furthermore, their first and third phases are the same. The second phase finds the mates for the leftmost left and the rightmost right parentheses of each $\log n$-size substring, but employs a different approach. Again, a binary matching tree with $n/\log n$ leaves is constructed. Each leaf has a single-node linked list
containing the first and last indices of the unmatched right parentheses (the right list); likewise, there is a left list for the unmatched left parentheses. At a parent node, the matching portions of the left list of the left child and that of the right list of the right child are “cut out” and stored. The right list of the left child is linked with the unmatched portion of the right list of the right child. Similarly, an updated left list is created at the parent. After the binary tree is constructed, the left list cut-outs stored at each internal node are concatenated (in some order) to obtain a final left list. Likewise, a final right list is formed. The matching left and right parentheses in these two final lists are aligned by using a linked list ranking algorithm on each of these lists. This is followed by actual matching of pairs. The splitting of the matching portions at each parent node must be performed in a constant time. For this purpose, a preprocessing stage computes two splitting pairs for each node. One of these pairs is the leftmost left parenthesis of the left subtree and its matching right parenthesis in the right subtree. Likewise, there is another pair for the rightmost right parenthesis of the right subtree. These splitting pairs are computed by using the tree navigation technique of the Bar-On and Vishkin algorithm, but a concurrent read is avoided by simply keeping the containing process trying to climb up from a right child. Diks and Rytter are unclear about how splitting pairs are employed to compute each parent node in O(1) time. This algorithm has some similarity to our Algorithms III and IV in that it has used intervals to represent reduced substrings, and it identifies the matching portions of the left and right children at each parent node, and stores them to be matched later. However, in addition to having numerous O(log n)-time stages, the Diks and Rytter algorithm uses linked lists as the underlying data structure, and employs a linked list ranking algorithm.

The Levcopoulos and Petersson algorithm [27] is unique in not relying on a binary matching tree. Instead, as a major tool, this algorithm has used Cole’s optimal sorting algorithm [10], which has a large constant of proportionality. First, a parallel prefix algorithm is employed to find nesting depths of each parenthesis. Next those parentheses whose nesting depths are multiples of log n are separated out, stable sorted, and matched in O(log n) time. These special parentheses partition the original string into several substrings, referred to as blocks. Each block is reduced through a series of processings. The unmatched parentheses in the reduced block are matched with those in the matching blocks. The boundaries of the matching block are easily identified by the two matching parentheses of those two special parentheses that define a block. Reduction of each block is carried out by first reducing the constituent subblocks of size log n using a processor each, and then reducing a group of log n consecutive subblocks using log n processors, and finally stable sorting those blocks that are large enough to contain more than one group. Each of these three steps requires O(log n) time. Reduction of a group requires pipelined scan by a processor assigned to a subblock through each of the subblocks to its right to first “book” and then match the right parentheses with the left parentheses of its own subblock. The two stage matching avoids any concurrent access, and pipelining ensures an O(log n) time bound.

The all tallest neighbors (ATN) problem consists of n vertical line segments S = \{s_i = (x_i, y_i) | 1 \leq i \leq n\}, such that x_i < x_{i+1} for 1 \leq i \leq n-1. Its solution consists of finding the left and the right tallest neighbors of each segment. The right tallest neighbor (RTN) of the i-th segment in set S is the j-th segment, i < j, if y_i < y_j and j is the minimum such integer (denoted by RTN_S(i) = j). If no such right neighbor exists in S, then RTN_S(i) = 0. The left tallest neighbor, LTN_S(i), is analogously defined. Although Kim [24] has solved this problem on the EREW PRAM in \(O(\log n)\) time employing \(\frac{n}{\log n}\) processors, Berkman et al. [5] have solved a related problem, called all nearest smaller values (ANSV) problem, on the CREW PRAM with same time and processor complexities, and on the CRCW PRAM using \(O(\log \log n)\) time and \(\frac{n}{\log \log n}\) processors. To solve parentheses-matching problem using an algorithm for the ATN problem, assign a -1 to each left parenthesis and 1 to each right parenthesis, and then find all prefix-sums, and create an instance of the ATN problem by replacing the i-th parenthesis with the ordered pair \((i, l)\), where l is its prefix sum. Now, for each left parenthesis, its RTN corresponds to its matching right parenthesis.

We sketch Kim’s \(O(\log n)\)-time, n-processor algorithm first. Then we discuss how it is extended to bring down the processor complexity to \(O(\frac{n}{\log n})\). A \(\sqrt{n}\) divide-and-conquer approach is used. The input set is partitioned into \(\sqrt{n}\) equal subsets, \(T_1, \ldots, T_{\sqrt{n}}\), and the ATN problem is recursively solved on each subset. Let \(R_i\) (or \(L_i\)) be the set of segments in partition \(T_i\) such that RTN_{\(T_i\)}(s) = 0 (or LTN_{\(T_i\)}(s) = 0). Let \(h_i\) be the tallest segment in the i-th partition, and let \(H = \{h_i | 1 \leq i \leq \sqrt{n}\}\). The ATN problem is next solved for the set \(H\). Let \(H_i\) consist of the segments \(h \in H\) with LTN_{\(H\)}(h) = \(h_i\), and the segment RTN_{\(H\)}(\(h_i\)), sorted by their indices. Let \(\Gamma^i\) be the set of segments of increasing height from \(L_i\) for each segment \(h\) in \(H_i\). Then, for each \(s \in R_i\), RTN_{\(S\)}(s) = \(t\), where \(t \in \Gamma_i\) is the shortest segment taller than \(s\) and can be obtained from the merged list \(R_i \cup \Gamma_i\). A total of \(\sqrt{n}\) mergings, one for each partition, can be shown to require \(O(\log n)\) time using \(n\) processors. Intricate processor assignments and replications must be carried out to avoid concurrent read and to ensure logarithmic time for these steps.

Now let us suppose that only \(\frac{n}{\log n}\) processors are available for the ATN problem. The major steps taken are as follows. The segments are stored into a doubly linked list, partitioned into \(\frac{n}{\log n}\) subsets, and the ATN problem is sequentially solved for each subset. The list \(H\) of the tallest segment of each subset is formed, and the ATN problem is solved on this list, using \(\log \frac{n}{\log n}\) processors and \(O(\log n)\) time using the previous algorithm. Then \(|H_i|\) copies of \(R_i\) are made, a processor is assigned to each such copy, and \(R_i\) is merged with an appropriate portion of \(\Gamma_i\). Calculation of individual \(|H_i|\), for \(1 \leq i \leq \frac{n}{\log n}\), needs parallel sorting.

In comparison with these algorithms, our algorithms use only arrays, and do not employ either pipelining or other involved techniques, including linked list ranking and parallel sorting. Furthermore, our algorithms consist of relatively
fewer steps, each implementable in a straightforward fashion.

VII. CONCLUSION

We have presented four polylogarithmic-time parallel algorithms for the parentheses-matching problem on the EREW PRAM model. They are presented in the order of successive improvements in the resource (i.e., time, processor, and working space) requirements. Algorithms I and II are very easy to implement, and Algorithm IV is the most sophisticated. Our algorithms also illustrate the development of, wherever necessary, suitable data structures, processor scheduling, and task partitioning schemes in a parallel environment.

Algorithms I and II are not cost-optimal, and the required processor-time product is $O(n \log n)$ as compared to $O(n)$ operations for a sequential algorithm, although there is a trade-off between their time and processor requirements. That is, Algorithm I runs in $O(n \log^2 n)$ time, employing $O(n \log n)$ processors, whereas Algorithm II requires $O(n \log n)$ time with $O(n)$ processors. On the other hand, both Algorithms III and IV are cost-optimal and are run in $O(n \log n)$ time, employing $O(n \log n)$ processors. Except for Algorithm III, all others make use of optimal (i.e., $O(n)$) working space. The continued innovation of such new algorithms for the parentheses-matching problem thus points to the richness in the structure of this important problem.

We point out that the set of balanced strings of parentheses is a context-free language, referred to as the one-sided Dyck language over one letter. In general, a one-sided Dyck language over $k$ letters, for $k \geq 1$, is the set of balanced strings of $k$ different kinds of parentheses [22]. For example, $\{\{\}()\}\} \} \} \}$ is a balanced string over three different kinds of parentheses. Ibarra et al. [23] have shown that one-sided Dyck languages over $k$ letters can be recognized in $O(n \log n)$ time using a polynomial number of processors. The algorithms presented here can easily be extended for strings of parentheses of more than one kind. Before matching a pair of left and right parentheses, an additional check is needed to ensure that the pair contains parentheses of the same kind.

Other than its natural applications in parsing, we have recently applied a parentheses-matching algorithm to designing cost-optimal parallel algorithms for breadth-first traversal of general trees and sorting a special class of integers [7], [9], and minimum coloring of interval graphs [13]. Anderson et al. [2] have employed a parentheses-matching algorithm to design parallel heuristics for bin-packing problems. Currently, we are investigating other classes of problems that can be efficiently solved in parallel, using parentheses-matching as a subalgorithm.

Recently, Mayr and Werchner [29] have presented an algorithm to match $n$ parentheses in $O(n \log n)$ time using $n$ processors on the hypercube model, which is a distributed-memory, message-passing parallel machine. In addition, they also show how their algorithm can be employed to determine if a string of parentheses of length $O(n \log n)$ is balanced in $O(n \log n)$ time using $n$ processors. Future work can be aimed at designing cost-optimal parallel algorithms for parentheses-matching on hypercube architectures. Such a result will imply, in light of our results in [7], [9], [13], that several other problems can possibly be solved cost-optimally on the hypercube computers.

ACKNOWLEDGMENT

We would like to thank S. Sahni and the anonymous referees for suggesting inclusion of some important references.

REFERENCES

IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 5, NO. 9, SEPTEMBER 1994


S. K. Prasad (S'86-M'91) received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1985, the M.S. degree in computer science from Washington State University, Pullman, in 1986, and the Ph.D. degree in computer science from the University of Central Florida, Orlando, in 1990.

Since 1990, he has been an Assistant Professor at Georgia State University, Atlanta. His current research interests are parallel algorithms and data structures, parallel discrete event simulation, parallel compilation, parallel graph algorithms, very large scale integration (VLSI) layout algorithms, neural networks, and computer interconnection networks.

Dr. Prasad has published more than 20 papers in journals and refereed proceedings in his areas of research interest.

S. K. Das (M'94) received the B.Tech. degree in computer science from Calcutta University, India, in 1983, the M.S. degree in computer science from the Indian Institute of Science, Bangalore, in 1984, and the Ph.D. degree in computer science from the University of Central Florida, Orlando, in 1988.

From 1988 to 1993, he was an Assistant Professor in the Department of Computer Science, University of North Texas, Denton, TX, USA, where he is currently a tenured Associate Professor and also a faculty member of the Center for Research in Parallel and Distributed Computing. His current research interests include design and analysis of parallel algorithms, parallel data structures, applied graph theory and combinatorics, and multiprocessor interconnection networks.

Dr. Das is a recipient of the Cambridge Nehru Scholarship in 1986 and an Honor Professor Award at the University of North Texas in 1991. In 1993, he received a scholarship to visit the Leonardo Fibonacci Institute in Trento, Italy. He serves on the Editorial Board of Parallel Processing Letters and Journal of Parallel Algorithms and Applications. He has been a member of program committees for several conferences. He has published more than 40 journal papers in his areas of research interest, and has presented his research work at many international and national conferences. He is a member of the IEEE Computer Society, ACM, the New York Academy of Sciences, and Sigma Xi.

C. C.-Y. Chen received the Ph.D. degree in computer science from the University of North Texas, Denton, TX, USA, in 1991.

He is currently an Associate Professor in the Department of Computer Science and Information Engineering, Tamkang University, Taiwan, Republic of China. He was previously with Xerox Corp. and IBM Corp., working in the areas of operating system support and performance evaluation. His research interests include design and analysis of parallel algorithms, distributed and parallel systems, interconnection networks, operating system support for multimedia application, and performance evaluation.

Dr. Chen is a member of ACM and the IEEE Computer Society.