Matching parentheses in parallel

Christos Levcopoulos and Ola Petersson

Department of Computer Science, Lund University, Box 118, S-22100 Lund, Sweden

Received 15 February 1990
Revised 1 July 1990

Abstract

Parallel algorithms for evaluating arithmetic expressions generally assume the computation tree form to be at hand. The computation tree form can be generated within the same resource bounds as the parenthesis matching problem can be solved. We provide a new cost optimal parallel algorithm for the latter problem, which runs in time $O(\log n)$ using $O(n/\log n)$ processors on an EREW PRAM. We also prove that the algorithm is the fastest possible independently of the number of processors available.

Keywords. Parenthesis matching, parallel algorithms, optimality, PRAM, arithmetic expression evaluation.

1. Introduction

The model of computation used in this paper is the least powerful among the parallel random access machines (PRAMS), namely the exclusive-read exclusive-write (EREW) PRAM. A PRAM employs $P$ synchronous processors all having access to a common memory. An EREW PRAM does not allow concurrent access to the same memory location. See [11] for a survey of PRAM results.

Let Seq($n$) be the fastest known worst-case running time of a sequential algorithm, where $n$ is the length of the input for the problem being considered. Since one processor can simulate a $P$ processor, $T$ time, parallel algorithm in time $O(PT)$, the best upper bound on the parallel time achievable using $P$ processors, without improving the sequential result, is of the form $O(Seq(n)/P)$. The cost of a parallel algorithm is defined as $PT$, and a parallel algorithm achieving the time bound...
O(\text{Seq}(n)/P) is said to be cost optimal. We also call an algorithm time optimal if it is the fastest possible.

The problem considered here is the parenthesis matching problem. Given a legal sequence of \( n \) parentheses, i.e., every parenthesis has its matching parenthesis in the sequence, for each parenthesis, find its matching parenthesis. Sequentially, the problem can trivially be solved in linear time by scanning the sequence forward once while maintaining a stack of the not yet matched left parentheses. Bar-On and Vishkin [3] and Sarkar and Deo [15] have given cost optimal \( O(\log n) \) time CREW PRAM algorithms by using \( O(n/\log n) \) processors. (In a CREW PRAM concurrent reads are allowed while concurrent writes are not.) Recently, Berkman et al. [5] gave a CRCW PRAM parenthesis matching algorithm which runs in time \( O(\log \log n) \) using an optimal number of processors, provided the nesting level of each parenthesis is known a priori. (In a CRCW PRAM concurrent reads as well as concurrent writes to the same memory location are allowed.) Their algorithm has been improved by Berkman and Vishkin [6] to run in time \( \alpha(n) \). However, it follows from Theorem 2.1 in this paper that computing the nesting levels takes \( \Omega(\log n/\log \log n) \) time.

Bar-On and Vishkin used the parenthesis matching as a subroutine in a parallel algorithm for generation of computation tree forms for general arithmetic expressions, which ran within the same resource bounds. Their algorithm was an improvement of Dekel and Sahni’s [9] EREW PRAM algorithm for generation of computation tree forms. The need for constructing computation tree forms in logarithmic time stems from that logarithmic time algorithms for arithmetic expression evaluation require the expression to be in that form. Cost optimal deterministic EREW PRAM algorithms for arithmetic expression evaluation, which run in time \( O(\log n) \) using \( O(n/\log n) \) processors, have been developed by Abrahamson, Dadoun, Kirkpatrick and Przytycka [11], among others. Observe that these results assume the computation tree form of the arithmetic expression to be at hand.

In this paper we propose an EREW PRAM algorithm for parenthesis matching which runs in time \( O(\log n) \) using \( O(n/\log n) \) processors, which is cost and time optimal. The CREW PRAM algorithm for the computation tree form generation of Bar-On and Vishkin made use of the concurrent reads only when matching parentheses. Therefore, a straightforward adoption of their method yield an EREW PRAM algorithm for generation of the computation tree form of an arithmetic expression which runs within the same resource bounds. Hence, our algorithm can be used as a proper preprocessing phase for the aforementioned algorithms for arithmetic expression evaluation. Furthermore, Sarkar and Deo [14] have shown that a class of block-structured languages can be parsed in logarithmic time on an \( O(n/\log n) \) processor EREW PRAM, given that parenthesis matching can be performed as efficiently. Independently, Anderson, Mayr and Warmuth [2], Diks and Rytter [10], and Tsang, Lam and Chin [16] have reported algorithms achieving the same upper bound as

\[ \alpha(n) \text{ is the inverse Ackermann function.} \]
ours. All these algorithms are completely different from the one presented in this paper, however.

Our \texttt{CREW PRAM} parenthesis matching algorithm, which is nonrecursive, starts by computing the depth, or nesting level, of each parenthesis. Then, we extract all parentheses whose depth is divisible by \( \log n \) and match them in parallel. These matched parentheses divide the whole sequence into consecutive subsequences, or blocks, within which the difference in depth is \( O(\log n) \). Next, we perform local matchings within the blocks, after which the total number of unmatched parentheses left are so few that they can be matched in a straightforward way. The algorithm uses both Cole's Parallel Merge Sort [7] and parallel prefix sums computation [12, 13] as subroutines.

The remainder of this paper is organized as follows. In Section 2 we provide time lower bounds for the parenthesis matching problem. In Section 3 a high-level description of the algorithm is given. In Section 4 we spell out the details and analyse our algorithm. Finally, in Section 5, the results are summarized.

2. Time lower bounds

In this section we derive time lower bounds for the parenthesis matching problem on both the \texttt{CREW PRAM} and the \texttt{PRIORITY CRCW PRAM} [4].

\textbf{Theorem 2.1.} Solving the parenthesis matching problem requires \( \Omega(\log n/\log \log n) \) time on a \texttt{PRIORITY CRCW PRAM} with a polynomial number of processors. On a \texttt{CREW PRAM}, \( \Omega(\log n) \) time is required independently of the number of processors available.

\textbf{Proof.} We make a reduction from the \texttt{Bitsum} problem, i.e., compute the sum of \( n \) bits. Let \( X = \langle x_1, \ldots, x_n \rangle \) be an instance of \texttt{Bitsum}. Construct the following sequence of parentheses:

\[
S(X) = ((\ldots ((s_1 s_2 \ldots s_n)\ldots)) ((\ldots ((\tilde{s}_1 \tilde{s}_2 \ldots \tilde{s}_n)\ldots)\ldots))
\]

where \( s_i \) is a left parenthesis if \( x_i = 0 \) and a right if \( x_i = 1 \), and \( \tilde{s}_i \) is a right parenthesis if \( x_i = 0 \) and a left if \( x_i = 1 \). It is easily verified that \( S(X) \) is a legal sequence of parentheses of length \( 8n + 6 \). It should be observed that the second half of \( S(X) \) is only due to making it a legal sequence.

Let \( \text{match}(i) \) denote the position of the parenthesis that matches the one in position \( i \). We claim that \( \text{Bitsum}(X) = n + 1 - \text{match}(4n + 3)/2 \), which is seen by observing:

1. If \( \text{Bitsum}(X) = 0 \), then \( \text{match}(4n + 3) = 2n + 2 \).
2. For each 1 in \( X \), \( \text{match}(4n + 3) \) decreases by 2.

Hence, the parenthesis matching problem is at least as hard as \texttt{Bitsum}. The lemma
now follows from the time lower bounds for \textsc{Bitsum} on a \textsc{priority crcw pram} by Beame and Håstad [4], and on a \textsc{crew pram} by Cook, Dwork and Reischuk [8]. □

Bar-On and Vishkin conjectured that \( \Omega(\log n) \) is a time lower bound on a \textsc{crew pram} if optimal speed-up is required. Theorem 2.1 proves an even stronger result. Note that the \textsc{crew pram} lower bound applies for the \textsc{erew pram} as well since the latter is a less powerful model of computation.

3. The algorithm

Let \( S = (s_1, \ldots, s_n) \) be a legal sequence of parentheses, and let \( |S| = n \) denote the \textit{length} of \( S \). We give a parallel algorithm which for each parenthesis finds its matching parenthesis in time \( O(\log n) \) by using \( n/\log n \) processors on an \textsc{erew pram}.

Like the algorithm in [3] we start by computing the depth, or nesting level, of each parenthesis. This is done in a preprocessing phase.

\textbf{Preprocessing Phase.}

For clarity, we use a temporal array \( A \) for indicating whether a parenthesis is a left or right one.

\begin{itemize}
\item \textit{Step 1.} Each processor scans its consecutive portion of \( \log n \) parentheses and sets \( A(i) = 1 \) if \( s_i \) is a left parenthesis, and \( A(i) = -1 \) otherwise.\(^2\)
\item \textit{Step 2.} Compute all prefix sums with respect to \( A(i) \), and store the result in the array \( \text{depth} \), i.e., \( \text{depth}(j) = \sum_{i=1}^{j} A(i) \), for \( 1 \leq j \leq n \).
\item \textit{Step 3.} If \( s_i \) is a right parenthesis, increment \( \text{depth}(i) \) by one.
\end{itemize}

\textbf{End of Preprocessing Phase.}

It should be clear that \( \text{depth}(s_i) \) now holds the depth of \( s_i \).\(^3\) Since the prefix sums computation can be implemented to run in time \( O(\log n) \) on an \( n/\log n \) processor \textsc{erew pram} [12, 13], the same holds for the whole Preprocessing Phase.

\textbf{Remark 3.1.} Given the depths of the parentheses, it is easy to determine whether the input sequence \( S \) is legal or not. \( S \) is legal if and only if \( \text{depth}(s_i) \geq 1 \), for \( 1 \leq i \leq n \), and \( s_n \) is a right parenthesis of depth one, which can be tested in \( O(\log n) \) time.

The aim of the next phase of the algorithm is to compute the array \( \text{match} \), such that \( \text{match}(i) = k \) and \( \text{match}(k) = i \) if and only if \( s_i \) and \( s_k \) are matching parentheses.

\(^2\) For simplicity, \( \log n \), which is to the base 2, is assumed to be an integer which divides \( n \).

\(^3\) We will use \( \text{depth}(s_i) \) synonymously with \( \text{depth}(i) \), indicating that it is actually the depth of a parenthesis, rather than the depth of a position.
Matching two parentheses thus means to write into the corresponding entries of the array match. Before proceeding with the next phase, however, let us make some useful observations that facilitate the understanding of the algorithm and simplify the analysis in the next section.

**Remark 3.2.** Let \( s_j \) be a left parenthesis and let \( s_k \) be its matching right parenthesis. Then, \( k = \min\{i | j < i \leq n \text{ and } depth(s_i) = depth(s_j)\} \). That is, a left parenthesis finds its matching right parenthesis at the closest following position containing a parenthesis of the same depth. The analogous result holds if \( s_j \) is a right parenthesis.

If the number of processors available is linear in the number of parentheses, the parenthesis matching problem has a simple solution:

**Lemma 3.3 (Submatch Lemma).** Let \( S \) be a legal sequence of \( n/\log n \) parentheses for which the depths are known. Then, \( S \) can be matched in \( O(\log n) \) time using \( n/\log n \) processors on an EREW PRAM.

**Proof.** Consider the following algorithm. First, make a stable sort of \( S \) with respect to the depths by applying Parallel Merge Sort [7]. A stable sort is obtained if the binary word for \( i \) is catenated at the end of \( depth(s_i) \), for each \( s_i \in S \). Since Parallel Merge Sort runs in \( O(\log n) \) time if \( n \) processors are available when sorting \( n \) elements, it sorts \( n/\log n \) elements in \( O(\log n) \) time by using \( n/\log n \) processors.

Second, assign one processor to each left parenthesis in the sorted sequence. Each processor matches its parenthesis with the parenthesis appearing in the next following position. This step is carried out in constant time and the correctness follows from the stability of the preceding sort and Remark 3.2. \( \square \)

The algorithm given in the last proof will be applied to solve smaller subproblems by our main algorithm. Let us therefore refer to it as **Submatch**.

Note that the depth of a parenthesis in a sequence of length \( n \) is an integer between 1 and \( n \). Hence, if there exists a stable integer sorting algorithm which runs in \( O(\log n) \) time by using \( n/\log n \) processors, the Preprocessing Phase together with algorithm Submatch, with the integer sort instead of Parallel Merge Sort, suggest a straightforward method for solving the parenthesis matching problem optimally. Unfortunately no such integer sort is known, even in stronger PRAM models than the EREW PRAM.

**Main Phase.**

**Step 1.** Let \( S_1 \) be the subsequence of \( S \) that consists of all \( s_j \) for which \( \log n \) divides \( depth(s_j) \). For each left parenthesis in \( S_1 \), check whether its right neighbour is a right parenthesis. If it is, match them.

**Step 2.** Denote by \( S_2 \) the subsequence of unmatched parentheses in \( S_1 \). Match \( S_2 \) by Submatch.
The parentheses in $S_2$ that are matched in Step 2 divide $S$ into a number of consecutive subsequences of unmatched parentheses (we ignore the parentheses matched in Step 1). We divide $S$ into blocks as follows. Consider two consecutive parentheses $s_j$ and $s_k$, $j < k$, in $S_2$. If $s_j$ and $s_k$ are both left parentheses, or both right parentheses, all parentheses between $s_j$ and $s_k$ in $S$ together with all parentheses between $s_{\text{match}(j)}$ and $s_{\text{match}(k)}$ in $S$ form a block. If $s_j$ and $s_k$ are not both left or right parentheses, we let all parentheses between them in $S$ form a block. It is clear that every parenthesis has its matching parenthesis in its own block. Further, the difference in depth between two parentheses within the same block is bounded by $2 \log n$. We proceed by performing local matchings within the blocks.

**Step 3.** Divide the blocks into subblocks of $\log n$ parentheses each. Allocate one processor per subblock and perform a sequential matching within each subblock.

**Step 4.** In each block, group together every $\log n$ subblocks. (The last group in a block may have less than $\log n$ subblocks.) Perform a local matching within each such group.

**Step 5.** Denote by $S_3$ the subsequence of $S$ consisting of all parentheses that have not yet been matched. Match $S_3$ by Submatch.

**End of Main Phase.**

4. Analysis and details

We argue for the correctness of the Main Phase and show that each step can be implemented to run in time $O(\log n)$ using $O(n/\log n)$ EREW PRAM processors.

**Step 1.** $S_1$ can easily be extracted from $S$ by first performing a parallel prefix computation with respect to whether the depth of each parenthesis is divisible by $\log n$ or not. The results of this computation are then used as addresses into the sequence $S_1$. The correctness of the matching follows by Remark 3.2. The step can be performed within the required resource bounds on an EREW PRAM.

**Step 2.** The extraction of $S_2$ from $S_1$ is done in the same way as in Step 1. To ensure that Submatch runs in $O(\log n)$ time we must show that $|S_2| = O(n/\log n)$.

Let $s_j$ and $s_k$, $j < k$, be two consecutive parentheses in $S_2$. The aim is to prove that for at least half of all such pairs, $k - j = \Omega(\log n)$. We distinguish two cases:

(a) $\text{depth}(s_j) \neq \text{depth}(s_k)$. Assume without loss of generality that $\text{depth}(s_j) < \text{depth}(s_k)$, i.e., $\text{depth}(s_k) \geq \text{depth}(s_j) + \log n$. Then $k - j \geq \log n$, since there must be at least one parenthesis of each depth, $\text{depth}(s_j) + 1, \ldots, \text{depth}(s_k) - 1$, between $s_j$ and $s_k$ in $S$.

(b) $\text{depth}(s_j) = \text{depth}(s_k)$.

*The division $\text{depth}(s_j)/\log n$ needs only be calculated for the first parenthesis handled by each processor. After that, the processor can determine whether each new parenthesis encountered belongs to $S_1$ by just adding or subtracting $1 \mod \log n$ depending on whether it is a left or right parenthesis.*
(b) Otherwise, if \( \text{depth}(s_j) = \text{depth}(s_k) \), we distinguish two subcases:

(i) \( s_j \) is a left and \( s_k \) is a right parenthesis. By Remark 3.2, \( s_j \) and \( s_k \) match each other. Since they were not matched in Step 1, there was another pair of parentheses \( s_l \) and \( s_m \), between \( s_j \) and \( s_k \), i.e., \( j < l < m < k \), in \( S_1 \), that was matched in Step 1. Thus, \( s_m \) belongs to the interval in \( S \) defined by \( s_j \) and \( s_k \), implying that \( \text{depth}(s_m) = \text{depth}(s_k) + \log n \). By applying the same argument as in case (a), we conclude \( k - j \geq k - m = \log n \).

(ii) \( s_j \) is a right and \( s_k \) is a left parenthesis. The number of such pairs is bounded by \( |S_2|/2 \), since the next consecutive pair in \( S_2 \), consisting of \( s_k \) and the parenthesis next to \( s_k \) in \( S_2 \), can obviously not be of the same kind.

Note that the subcases where \( s_j \) and \( s_k \) are both left or right parentheses cannot appear, since they would contradict Remark 3.2.

We conclude that there are at least \( |S_2|/2 \) consecutive pairs of parentheses in \( S_2 \) which are at least \( \log n \) positions apart in \( S \). Hence, \( |S_2| \leq 2n/\log n \), which completes the analysis of this step.

**Step 3.** Straightforward.

**Step 4.** Rather than giving all the tedious details, we just sketch how this step can be performed. Observe that after Step 3, each subblock consists of a series of right parentheses followed by left parentheses, i.e., \( \ldots \rangle \ldots \langle \ldots \). Recall that a subblock consists of at most \( \log n \) parentheses and that we have one processor per subblock. To avoid read conflicts we proceed in two stages. In the first stage, which is pipelined, each processor finds a number of series of right parentheses which will match its own left parentheses; second, the actual matching is performed. Let \( B_i \) and \( p_i \), \( 1 \leq i \leq \log n \), denote the subblocks and processors associated to subblocks within a group, respectively.

**Stage 1:**

\[
\text{for } j := 1 \text{ to } \log n - 1 \text{ do}
\]

\[
\begin{align*}
\text{(a) Processor } p_i, & \text{ checks if there are any unmatched right parentheses left in } B_{i+j}. \text{ If there are enough to match all unmatched left parentheses in } B_i \text{ then } p_i \text{ "books" as many as it needs, starting with the leftmost (right) parenthesis in } B_{i+j}. \text{ Otherwise it books as many as there are available;} \\
\text{(b) } p_i & \text{ updates information regarding how many unmatched left parentheses it has and how many unmatched right parentheses there are left in } B_{i+j};
\end{align*}
\]

**Comment:** The booking in (a) is performed by letting pointers indicate where the first and last matching parentheses of a series are located. This can be done in constant time if we keep the appropriate overhead information.
Stage 2:
Perform the actual matching of the booked parentheses;

It should be clear that both stages can be performed in logarithmic time and that no concurrent reads occur. The correctness can be proved by induction on \( j \).

Step 5. Extracting \( S_3 \) from \( S \) is done in the same way as in Step 1. As in Step 2 we have to prove that \( |S_3| = O(n/\log n) \) for the Submatch Lemma to apply. Observe that all blocks of length at most \( \log^2 n \) were matched completely in Steps 3 and 4. Hence, only blocks of larger length might contribute to \( S_3 \). We claim that the number of unmatched parentheses in such blocks decreased by the factor \( \Omega(\log n) \) during Steps 3 and 4.

Consider a block having \( k > \log^2 n \) unmatched parentheses before Step 3. In Step 4, at most \( \lceil k/\log^2 n \rceil \) groups are formed in this block, and within each group all matchings are performed. Since the difference in depth within a block, and thereby a group, is bounded by \( 2\log n \), at most \( 4\log n \) parentheses in each group might remain unmatched after Step 4. Hence, the number of remaining unmatched parentheses in this block is bounded from above by \( 4\log n \cdot \lceil k/\log^2 n \rceil \), that is, \( k/\Omega(\log n) \).

Now, it follows that \( |S_3| = O(n/\log n) \) since all blocks contributing to \( S_3 \) have decreased in length by the factor \( \Omega(\log n) \), and their original total length was at most \( n \).

Theorem 2.1 combined with the above discussion and analysis implies

**Theorem 4.1.** There is an \( O(n/\log n) \) processor \( \text{EREW PRAM} \) algorithm which solves the parenthesis matching problem in time \( O(\log n) \), which is cost and time optimal.

By adopting the method of Bar-On and Vishkin [3] for generating the computation tree of a general arithmetic expression, by using parenthesis matching, a consequence of Theorem 4.1 is

**Corollary 4.2.** Given a general arithmetic expression of length \( n \), its computation tree can be constructed in time \( O(\log n) \) using \( O(n/\log n) \) processors on an \( \text{EREW PRAM} \).

5. Conclusion

We have presented a new parallel algorithm for the parenthesis matching problem. The algorithm runs in time \( O(\log n) \) using \( O(n/\log n) \) processors on an \( \text{EREW PRAM} \), which is cost and time optimal. Basically, the algorithm uses the depths of the parentheses to divide the sequence into subsequences. These subsequences are then processed independently, after which there are so few matchings left that they can be performed in a straightforward way.
The algorithm has applications in the generation of computation tree forms of general arithmetic expressions and in parsing of certain block-structured languages.

Acknowledgement

The authors thank the anonymous referee for helpful comments. We would also like to thank Arne Andersson for suggestions that improved the presentation of the algorithm.

References