An Optimal Parallel Algorithm for Arithmetic Expression Parsing

Weian Deng & S. Sitharama Iyengar
Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803

Abstract

We present an optimal expression parsing algorithm using SIMD-SM EREW model of computation, with a time complexity of $O(\sqrt{n})$ using $\sqrt{n}$ processors.

1: Introduction

Parallelized arithmetic expression parsing has been intensively studied recently. Many papers have been written on this topic. [2], [3] and [5] are some of the important results. Dekel and Sahni [3] considered the generation of postfix form from infix form for a given arithmetic expression of length $n$, and the translation of postfix form into tree form. Originating from the standard priority-based sequential infix to postfix algorithm, using stable sort technique to pair the matching parentheses together to exchange information among the tokens in an expression in parallel, they came up with two parallel algorithms on an SIMD-SM model. One used $n$ processors and $O(\log n)$ time, and the other used $n^2 / \log n$ processors and $O(\log n)$ time. Bar-On and Vishkin [2] presented the best parallel algorithm for obtaining the tree form of an arithmetic expression. They came up with an optimal parallel algorithm with $O(\log n)$ time complexity using $n / \log n$ processors. The heart of their algorithm consists of the development of an efficient parenthesis pairing algorithm in $O(\log n)$ time with $n / \log n$ processors, and the introduction of the concept of simple expressions. However, their algorithm is based on the SIMD-SM CREW model which is more powerful than that used by Dekel and Sahni. [3] Srikant [5] introduced another parallel algorithm for expression parsing. But his focus was on developing algorithms running on the other kinds of computational models, like mesh-connected machine, and cube-connected machine, etc.

In this paper, we discuss an optimal parallel algorithms for tree form generation of arithmetic expressions on SIMD-SM EREW model. The main idea here is how to avoid the read conflict posted by Bar-On and Vishkin’s algorithm by modifying their parenthesis pairing algorithm. In next section, we will introduce some necessary concepts and notations. In section 3, an optimal parenthesis pairing algorithms on SIMD-SM EREW model are presented.

2: Preliminaries

The problem we concern here is how to construct a binary evaluation tree for a proper arithmetic expression of length $n$. The expression is constructed from constants, variables, parentheses and operators (+, -, *, / and ^). The operators follow the usual priority rule and associativities. All operators except ^ are left associative. Each proper arithmetic expression corresponds to exactly one binary tree representation, in which all the leaves correspond to the operands and all the internal nodes correspond to the operators of the expression. The connection between these nodes are determined by the priorities and the associativities.

Definition 1: A linear expression is an proper arithmetic expression which consists of constants, variables and a set of operators with the same priority.

Definition 2: A sub-expression is a successive portion of a given arithmetic expression enclosed by a pair of matching parentheses.

Definition 3: A simple expression is an expression in which all operators within one sub-expression but not in any other sub-expression are of the same priority. [2]

In other words, a simple expression is an arithmetic expression in which each sub-expression of it is a linear expression, provided that all the sub-expressions within...
the former mentioned sub-expression are considered as its
operands. Therefore, any sub-expression of a simple
expression can be seen as a linear expression as well as
an operand for its upper level sub-expression. As an operand,
each sub-expression can be represented by its root
operator. In this way, the complicated tree form for an
expression can be found by simply translating the
expression to a simple expression, then finding all the tree
expression can be represented by its root
expression can be found by simply translating the
expression to a simple expression, then finding all the tree
forms for all the sub-expressions in the simple expression.

Here, we adopt the same strategy as Bar-On and Vishkin. The difference is the parenthesis pairing method.
So, in the rest of the paper, we discuss parenthesis pairing
algorithms only. The whole algorithm for expression
details about the whole algorithm, see [6].

3: Parentheses pairing

The problem of parenthesis pairing is that, given a
proper parenthesis sequence of length n, n is an even
number, pair all the matching left parentheses with their
right parentheses.

Suppose the input sequence is divided into m
segments, where n = mq, for a certain positive integer q.
Each segment has q parentheses. These segments are not
necessary to be the proper ones. In each segment \( S_k \), 1 ≤
k ≤ m, there are two kinds of parentheses; one includes all
the parentheses which can be paired within the segment,
and the other includes the parentheses in that segment
unable to be paired. Removing all the paired parentheses
from the segment, we can get a sequence of the form:

\[
) \ldots ) ( \ldots ( ( i \text{ successive right parentheses are followed by } j \\
\text{successive left parentheses.}
\]

**Definition 4:** The sequence of parentheses formed by
removing from a successive segment \( S_k \) of a proper
parenthesis sequence all the paired parentheses within
the segment, is called a unpaired sequence in segment
\( S_k \) and denoted by \( U_k \).

If matching parentheses have been found for all the left
most left parenthesis (LMLP in short) and the right most
right parentheses (RMRP in short) of all the unpaired
sequences of a given proper sequence, and these paired
parentheses are marked, there exists a very useful property.

[2]

**Property 1:** In every unpaired sequence \( U_k \), 1 ≤ k ≤ m
of a given proper parenthesis sequence, the sequence
of left parentheses to the right of the LMLP and to
the left of next marked left parenthesis can be paired
one after the other by the sequence of right
parentheses to the right of the corresponding right
parenthesis of this LMLP.

[2] has the proof for this property.

Let the proper parenthesis sequence input have length
n, N processors are used. The input is divided into \( \left\lfloor n/N \right\rfloor \) segments \( S_k \), 1 ≤ k ≤ \( \left\lfloor n/N \right\rfloor \). \( U_k \) is the unpaired
sequence of \( S_k \). For each sub-expression in the input \( E(j) \), 1 ≤ j ≤ n , a function is assigned as:

\[
\text{level}(j) = \begin{cases} 
\text{the number of } '(' \text{ to its left} & \text{if } E(j) = '\right)' \\
\text{the number of } '(' \text{ to its left} & \text{if } E(j) = '\right)'
\end{cases}
\]

**Property 2:** The matching parenthesis of LMLP \( k \) in \( U_k \)
is within the first unpaired sequence \( U \) to its right
with level(LMLP) ≥ level(RMRP). The matching
parenthesis of RMRP \( k \) in \( U_k \) is within the first
unpaired sequence \( U \) to its left with level(RMRP)
≥ level(LMLP).

**Proof:** We prove the LMLP \( k \) case only. For RMRP \( k \),
the proof is similar.

It is not possible for the matching parenthesis of
LMLP \( k \) to be within one of the unpaired sequences to
its left because this matching parenthesis should be to
the left of LMLP \( k \). Nor can it locate within \( U \).

Suppose there exists a positive integer \( j \), \( k < j \leq l \)
such that the matching right parenthesis \( R_k \) of
LMLP \( k \) is within \( U \). Then we have level( \( R_k \) ) =
level( LMLP \( k \) ) Because level( \( R_k \) ) ≥ level ( LMLP )
Therefore level( LMLP \( k \) ) ≥ level ( RMRP )
This is contradictory to the fact that \( l \) is the first
unpaired sequence to the right of \( U \) satisfying level( LMLP \( k \) ) ≥ level( RMRP )
So, no such \( j \) exists.

Because level( \( LMLP \) \( k \) ) ≥ 1 and the right most
right parenthesis of a proper parenthesis sequence has
a level value of 1, a segment satisfying the given
condition can be found for each LMLP.

The matching right parenthesis must be within \( U \)
because level( \( LMLP \) \( k \) ) ≥ level( RMRP \( l \) ) and
level( LMLP \( k \) ) ≤ level( RMRP \( l \) ) So, in the
sequence of the right parentheses in \( U \), there must be
one parenthesis (say \( R_k \) ) such that level( \( R_k \) ) =
level ( LMLP \( k \) ).
From this property, an new algorithm can be found. It finds the matching parentheses for LMLPs and RMRPs by allocating the segments where the matching parentheses are allocated and then searches through the corresponding segments for the matching parentheses. This is given below.

Algorithm (SIMD-SM EREW)

1. Partition the input into \( N \) successive segments of length \( n/N \), and assign each segment a processor. In parallel, all the processors perform sequential pairing algorithm to find all the paired parentheses within each segment, and remove them by marking. The result is \( N \) unpaired sequences.

2. Assign each parenthesis a nesting level value by the method described by Dekel and Sahni [3].

3. Allocate the matching unpaired sequences for all the LMLP\(_k\), \( 1 \leq k \leq N \). In parallel, each processor \( k \) associated with LMLP\(_k\) looks for the RMRP in the next unpaired sequence \( U_{k+1} \) then the second next \( U_{k+2} \) and so on, until an unpaired sequence \( U_l \) is allocated which satisfies that level( LMLP\(_k\) ) \( \geq \) level( RMRP\(_l\) ). A similar approach is also applied to RMRPs.

4. Each processor searches its unpaired sequence \( U_l \) found in step 3 for the matching right parenthesis. This is done by a linear search on \( U_l \) looking for the right parenthesis with the same level value. The paired parentheses are then marked. To avoid read conflict, a pipeline approach is used: \( P_1 \) begins first. When \( P_1 \) is searching the second parenthesis, \( P_2 \) begins, etc. A similar approach is also applied to RMRPs.

5. In parallel, each processor matches its left parentheses from the LMLP to the left until it meets a left parenthesis marked in step 4, or until all the left parentheses in this unpaired sequence are finished. A similar approach is also applied to RMRPs.

For example, let’s consider a proper input of

\[
\begin{array}{c}
( ( ( ) ( ( ) ) ( ( ) ) ) )
\end{array}
\]

with length of 16, Suppose 4 processors are used. In step 1, input is divided into four segments and results in four unpaired sequences

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
4 & 3 & 3 & 4 \\
4 & 3 & 2 & 1
\end{array}
\]

with the number below as the level value assigned to each parenthesis and the number above as segment number. In step 3, the segments at which the matching parentheses of the LMLPs are allocated, are found first. Then for a similar process is applied to the RMRPs. After this step, we get

\[
\begin{array}{cccc}
1 [4,0] & 2 [4,1] & 3 [4,2] & 4 [0,1] \\
1 & 2 & 2 & 3 \\
4 & 3 & 3 & 4 \\
4 & 3 & 2 & 1
\end{array}
\]

The two numbers within the braces are the matching segments for LMLP and RMRP respectively. In step 4, the matching parentheses for LMLP and RMRP are found and the paired parentheses are marked

\[
\begin{array}{cccc}
# & # & # & # \\
( ( ( ) ( ( ) ) ( ( ) ) ) ) \\
1 & 2 & 2 & 3 \\
4 & 3 & 3 & 4 \\
4 & 3 & 2 & 1
\end{array}
\]

Finally, step 5 pairs all the rest of the parentheses.

The correctness and effectiveness of algorithm 3 is given in theorem 3 below.

**Theorem 3:** Algorithm 3 can pair the matching parentheses for a given proper parenthesis sequence in time \( t = O(n/N + N) \) with an optimal cost \( c = O(n) \), for \( N \leq \sqrt{n} \).

**Proof:** The correctness of algorithm is guaranteed by the two properties of unpaired sequences. Step 1 and Step 2 are obviously correct. (See [2] and [3]). The correctness of Step 3 and Step 4 is given by property 2. From property 1, we can derive that step 5 is true.

Regarding the time complexity and cost:

<table>
<thead>
<tr>
<th>Step</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( O(n/N) )</td>
</tr>
<tr>
<td>2.</td>
<td>( O(n/N) + O(\log N) )</td>
</tr>
<tr>
<td>3.</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>4.</td>
<td>( O(n/N) )</td>
</tr>
<tr>
<td>5.</td>
<td>( O(n/N) )</td>
</tr>
</tbody>
</table>

So, totally \( t = O(n/N + N) + O(\log N) + O(n) + O(n/N) + O(n/N) = O(n/N + N) \)
\[ c = N \times O(n/N + N) = O(n + N^2) \]

To make algorithm 3 a cost optimal one, the condition is

\[ n \geq N^2 \]

So,

\[ N \leq \sqrt{n} \]

5.0 Conclusion

We have presented an optimal algorithm for arithmetic expression parsing in SIMD-SM-EREW model. Our algorithm employs pipeline approach to achieve optimal performance. Finally, a comparison of our algorithms with the other existing algorithms is given in Table 1.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>N</th>
<th>Model</th>
<th>Time</th>
<th>Cost = Time × N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekel &amp; Sahni (1983)</td>
<td>( n )</td>
<td>SIMD-SM</td>
<td>( O(\log^2 n) )</td>
<td>( O(n\log^2 n) )</td>
</tr>
<tr>
<td>Bar-On &amp; Vishkin (1985)</td>
<td>( n/\log n )</td>
<td>SIMD-SM-R</td>
<td>( O(\log n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Srikant (1990)</td>
<td>( n )</td>
<td>Cube-connected</td>
<td>( O(\log^2 n) )</td>
<td>( O(n\log^2 n) )</td>
</tr>
<tr>
<td>Deng &amp; Iyengar (1991)</td>
<td>( \sqrt{n} )</td>
<td>SIMD-SM</td>
<td>( O(\sqrt{n}) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

4: Reference


