Parallel Parsing of Arithmetic Expressions

Y. N. SRIKANT

Abstract—New parallel algorithms for parsing arithmetic expressions on mesh-connected, shuffle-connected, cube-connected, and cube-connected cycles models of parallel computation are presented. On the mesh-connected computer, the algorithm requires n processors and $O(\sqrt{n})$ time; on the other models, n processors and $O(\log^2 n)$ time are required.

Index Terms—**Arithmetic expressions, parallel processing, parsing.**

Manuscript received March 15, 1987; revised September 20, 1988.

The author is with the Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.

IEEE Log Number 8930175.

I. Introduction

Parsing arithmetic expressions on parallel computers has created much interest recently [2], [3], [9]. But all these efforts have been to develop parallel algorithms for parsing expressions on a parallel random access machine model of computation with shared memory (PRAM).

In this paper, we present parallel algorithms for parsing expressions on mesh, shuffle, cube, and cube-connected cycles parallel computers [1], [4]. With n processors, it requires $O(\sqrt{n})$ time on the mesh-connected model and $O(\log^2 n)$ time on others. For the mesh-connected computer, we use wrap-around and normal row-major ordering. For the shuffle computer, we use an extra connection between adjacent processors and thus four connections per processor are required. For details of these and other models, the reader is referred to [1], [4], [6]–[8].

II. The Parallel Algorithm

We permit arithmetic expressions with the operators $+$, $-$, $*$, $/$, and $\uparrow$, parentheses, constants, and variables with the usual priorities. All operators except $\uparrow$ are assumed to be left associative.

The parallel parsing algorithm consists of four major steps:

1) Parenthesize the given expression fully so that each subexpression within a pair of parentheses has operands connected by a pair of operators of exactly the same priority.

2) Delete redundant parentheses.

3) Separate the subexpressions at each level of parenthesis nesting and determine the root of the tree form of each subexpression in parallel.

4) Separate the subexpression at each level of parenthesis nesting and determine the children of each operator in the tree form of each subexpression in parallel.

We now explain these steps in detail. The implementation of each step on the different models of computation is examined in Section III. We assume that each element of the expression has been assigned one processor.

Definition: A simple expression (se) is an expression in which all operators not enclosed by parentheses are of the same priority. For example, $a + (b/c) - d$ is an se.

We insert parentheses in the expression so that all subexpressions in it, at all levels of parenthesis nesting, become se's.

Step 1: The rules for parenthesizing a given expression are given below [2]:

1) For each $+$, $-$ operator insert two left (right, resp.) parentheses to its right (left, resp.).

2) For each $*$, $/$ operator, insert one left (right, resp.) parenthesis to its right (left, resp.).

3) For each left (right, resp.) parenthesis, insert two additional left (right, resp.) parentheses to its right (left, resp.). In addition, three left (right, resp.) parentheses are added at the beginning (end, resp.). After the completion of this step, the whole expression is enclosed in a pair of parentheses. Redundant parentheses are deleted in the next step.

Since a maximum of five parentheses are inserted per element, these may be stored within each processor in its scratch memory. Insertion simply requires inspecting the element contained in each processor.

After this step, the elements of the expression including the newly inserted parentheses are renumbered to get an index for each element. Whenever an element is moved to another processor, this index is also moved along with it. We call this index as OI (original index). From now on we refer to left (right, resp.) parenthesis as lp (rp, resp.).

Step 2:

a) Extract only the parentheses by packing.

b) Compute nesting levels for all parentheses i as $\mathrm{NL1}(i)$ = (number of left parentheses in the range 1 to i) $-$ (number of right parentheses in the range 1 to $(i - 1)$).

c) Sort the parentheses (only) using NL1. Now the matching lp's and rp's come together and let them exchange information about their OI's. A stable sort is needed.

d) Distribute the parentheses to the processors indicated by their OI's.

The above steps are required to inform each lp (rp, resp.) about its matching rp (lp, resp.).

e) Every lp (rp, resp.) marks itself as "deleted" if the element to its immediate right (left, resp.) is an lp (rp, resp.) and the matching rp's (lp's, resp.) of these two lp's (rp's, resp.) are also adjacent [2]. Furthermore, if the index of the matching rp (lp, resp.) is (self index $+2$) (or (self index -2), resp.), i.e., only one variable or constant is enclosed between a pair of parentheses, then the corresponding lp and rp can be marked as "deleted."

f) Remove the parentheses marked "deleted" by packing and renumber the elements. The new index obtained is the new OI.

Step 3: Now we proceed to determine the root of the tree form of each subexpression at each nesting level of parentheses in parallel.

a) Compute nesting levels $\mathrm{NL2}(i)$ for all symbols i (not just parentheses) as: $\mathrm{NL2}(i)$ = (number of lp's in the range 1 to i) $-$ (number of rp's in the range 1 to $(i - 1)$). NL2 is useful in separating subexpressions.

b) For an element, if the left neighbor is an lp or the right neighbor is an rp, then mark it as a "border" element. Both these conditions cannot be simultaneously true for any element since redundant parentheses have been deleted.

c) Sort the symbols using NL2 as the key, and separate subexpressions at each nesting level of parentheses.

d) In the case of an se, either the first or the last operator in the expression is the root of the tree form of the expression depending on whether the operators are right or left associative, respectively (this should be intuitively obvious; for proofs see [9]). After step 3c) the first (last) operator will be either next or second next on the right (left, resp.) to the lp (rp, resp.) (this is because we have moved parentheses along with the symbols they enclose). Thus, by looking at two symbols on the left or right, the rp's and lp's can determine whether the first or the last operator is the root.

e) Extract only the parentheses by packing and bring together the matching lp's and rp's of each subexpression. The lp and the rp now exchange information about the root of the tree form of the subexpression they enclose and also the "border" markings of themselves.

f) Distribute the parentheses and the elements back to their original places indicated by OI. Now each lp and rp has information about the root of the tree form of the subexpression they enclose.

Step 4: All that remains is the computation of the children of the operators in the expression. The constants and variables do not have any children since they are leaves.

a) Compute nesting levels $\mathrm{NL3}(i)$ for all symbols i as: $\mathrm{NL3}(i)$ = (number of lp's in the range 1 to $(i - 1)$) $-$ (number of rp's in the range 1 to i). Note that NL3 is different from NL2.

b) Sort the symbols using NL3 as the key, and separate the subexpressions at each nesting level. Now the operands which are parenthesized subexpressions are represented by the parentheses only. It should be noted that the roots of the tree forms of the parenthesized subexpressions are available in the parentheses and these roots are the only information required for determining the children of the operators (for details see [9]). This is because no other element except the root of the tree form of a parenthesized subexpression can form the child of any operator outside the subexpression.

c) *Determination of the left child of the operators:*

i) For a right associative operator, the left child is the root of the tree form of the subexpression on the left; this is either the constant or id on the left or the root of the tree form of the parenthesized subexpression on the left.

ii) For a left associative operator, the left child is the *operator* to its left. If the operator under consideration is itself the leftmost operator (which is the case if the element to the left of the operator is a "border" element), then the root of the tree form of the subexpression on the left (which may be a constant, a variable or a

Given expression: $a / b + c \uparrow (d / e + f \uparrow g)$

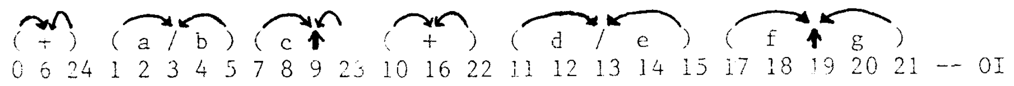
After step 1 (insertion of parentheses) we get,

$((\underline{(}a\underline{)} / \underline{(}b)\underline{)}) + \underline{(}(c \uparrow ((\underline{(}d\underline{)} / \underline{(}e)\underline{)}) + \underline{(}(f \uparrow g)\underline{)}))\underline{)})$

After step 2 (deletion of redundant parentheses) the underlined parentheses get deleted, and the expression becomes,

```
( ( a / b ) + ( c ↑ (  (  d  /  e  )  +  (  f  ↑  g  )  )  )  )
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 -- OI
```

After steps 3(i), 3(ii), and 3(iii) we obtain



```
( + )    ( a / b ) ( c ↑      ( +  )  (  d  /  e  )  (  f  ↑  g  )
0 6 24 1 2 3 4 5 7 8 9 23 10 16 22 11 12 13 14 15 17 18 19 20 21 -- OI
```

from which the roots of the tree forms of the subexpressions at each nesting level are determined and stored in the parentheses. These are shown by arrows. After steps 4(i) and 4(ii) we obtain,

```
( )   ( ) + ( )   a / b c ↑ (  )   (  )  +  (  )   d  /  e  f  ↑  g
0 24  1 5 6 7 23  2 3 4 8 9 10 22  11 15 16 17 21  12 13 14 18 19 20
 0        1            2                3                 4        -- NL3
```

from which the tree given below is obtained.

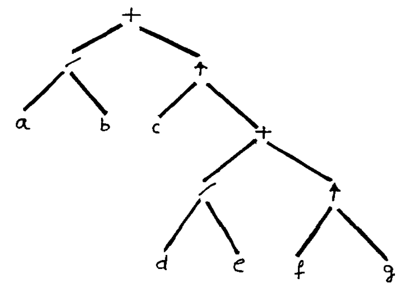


Fig. 1.

parenthesized subexpression, see i) above) is the left child; otherwise the index of the operator on the left is determined as either (index of the constant or variable to the left of the operator $-$ 1) or (index of the matching lp of the rp to the left of the operator $-$ 1), as the case may be.

d) *Determination of the right child of the operators:* Similar to step 4c) above (for details see [9]).

e) Distribute all the elements with their left and right information to their original places using OI.

Fig. 1 shows a detailed example.

III. Complexity Analysis

Table I below summarizes the various computations used in the algorithm and their time complexity. For details, the reader is referred to [1], [5]-[8].

Thus, using only n processors, the total time required for computing the tree form of an arithmetic expression is no more than $O(\sqrt{n})$ on the mesh-connected computer and $O(\log^2 n)$ on other models of computation.

TABLE I

Step Number	Computation	Main Technique	Time complexity with n processors on mesh-connected model	Time complexity with n processors on others
1	parentheses insertion	self-inspection	$O(1)$	$O(1)$
2	nesting levels	partial sum	$O(\sqrt{n})$	$O(\log n)$
2	checking neighbors	left/right shift	$O(1)$	$O(\log n)$*
2	packing	sorting	$O(\sqrt{n})$	$O(\log^2 n)$
2	distribution	sorting	$O(\sqrt{n})$	$O(\log^2 n)$
2	sorting	---	$O(\sqrt{n})$	$O(\log^2 n)$
2	renumbering elements	partial sum	$O(\sqrt{n})$	$O(\log^2 n)$
3,4	same as in step 2 above	---	$O(\sqrt{n})$	$O(\log^2 n)$

* Shift and pack operations require $O(1)$ and $O(\log n)$ time on our model of shuffle computer (see [8]).

References

[1] S. G. Akl, *Parallel Sorting Algorithms.* New York: Academic, 1985.

[2] I. Bar-On and U. Vishkin, "Optimal parallel generation of a computation tree form," *ACM Trans. Programming Languages Syst.*, pp. 348-357, Apr. 1985.

[3] E. Dekel and S. Sahni, "Parallel generation of postfix and tree forms," *ACM Trans. Programming Languages Syst.*, pp. 300-317, July 1983.

[4] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

[5] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. Comput.*, vol. C-30, pp. 101-107, Feb. 1981.

[6] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458-473, May 1977.

[7] F. P. Preparata and J. E. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," *Commun. ACM*, pp. 300-309, May 1981.

[8] J. T. Schwartz, "Ultracomputers," *ACM Trans. Programming Languages Syst.*, pp. 484-521, Oct. 1980.

[9] Y. N. Srikant and P. Shankar, "A new parallel algorithm for parsing arithmetic infix expressions," *Parallel Comput.*, vol. 4, pp. 291-304, 1987.