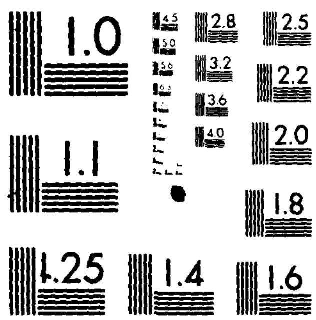
AD-A116 290 MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE F/G 12/1
PARALLEL GENERATION OF POSTFIX AND TREE FORMS.(U)
APR 81 E DEKEL, S SAHNI N00014-80-C-0650
UNCLASSIFIED TR-81-4 NL

END
DATE FILMED
7-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

R-81-4

0

AD A116290

DTIC FILE COPY

DTIC
ELECTE
JUN 29 1982
S
E

Computer Science Department
136 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

Parallel Generation of Postfix and Tree Forms

by

Eliezer Dekel and Sartaj Sahni

Technical Report 81-4
April 1981

Cover design courtesy of Ruth and Jay Leavitt

Parallel Generation of Postfix and Tree Forms*

Eliezer Dekel and Sartaj Sahni
University of Minnesota

Abstract

Efficient parallel algorithms to obtain the postfix and tree forms of an infix arithmetic expression are developed. The shared memory model of parallel computing is used.

Key Words and Phrases: Arithmetic expressions, postfix, infix, tree form, parallel computing, complexity.

*This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

1. Intorduction

Much work has been done on the parallel evaluation of arithmetic expressions. Some references are [1], [2], [8], [10], and [11]. Brent [1], for instance, has shown that arithmetic expression containing n, $n \geq 1$, operands; operators (+, *, and /); and parenthesis can be evaluated in $4\log_2 n+10(n-1)/p$ time when p processors are available. Unfortunately, little work seems to have been done on the parallel generation of executable code for arithmetic expressions. Fischer [5] considers the parsing of expressions on a vector (pipelined) machine. No work has been done on the parsing of expressions on parallel multiprocessor machines. In this paper, we address this problem. Specifically, we study the following problems:

(1) parallel generation of the postfix form
(2) parallel generation of the binary tree form

In both cases, we start with the infix form of the expression. Further, we assume that the input infix expression is syntactically correct. The reader unfamiliar with the postfix and tree forms of an expression is referred to Horowitz and Sahni [6].

The study of the two problems cited above is motivated by the following considerations:

(1) We could conceivably build a special purpose infix-to-postfix chip that could be used like a peripheral on a very high speed number cruncher. The use of this parallel translator chip would speed compilation of programs.
(2) Most code optimizers for single processor machines start with the tree form of an expression. Hence a high speed special purpose chip that performs the translation from infix to tree form could be used in the context of (1).
(3) If the program is to be executed on a parallel machine, it can also be compiled on that machine using a parallel compiler. Such a compiler will need to be able to translate in parallel, from the infix form to a more usable form. The postfix and tree forms are two such forms. In fact, the parallel evaluation methods suggested in [1], [2], [8], [10], and [11] all begin with the tree form of the arithmetic expression.
(4) While the length of individual arithmetic expressions in typical programs is small (Knuth [7]), Kuck [9] has shown that optimizing compilers for parallel machines can generate very long expressions even when the input program contains only short expressions. Furthermore, it is possible to view the entire program as a single expression and obtain its postfix form.

The model of parallel computation that we shall use here is commonly referred to as the shared memory model (SMM). This has the following characteristics:

(1) There are p processing elements (PEs) or processors. These are indexed $0,1,\ldots,p-1$ and an individual PE may be referenced as in PE(i). Each PE is capable of performing the standard arithmetic and logical operations. In addition, each PE knows its index.
(2) There is a common memory that is shared by all the PEs. All p PEs can read and write into this memory in the same time instance. If two PEs attempt to read the same word of memory simultaneously, a read conflict occurs. Similarly, if two PEs attempt to simultaneously write into the same word of memory, a write conflict occurs. In this paper, we assume that read and write conflicts are prohibited.
(3) The PEs are synchronized and operate under the control of a single instruction stream.
(4) An enable/disable mask can be used to select a subset of the PEs that are to perform an instruction. Only the enabled PEs will perform the instruction. The remaining PEs will be idle. All enabled PEs execute the same instruction. The set of enabled PEs can change from instruction to instruction .

Much work has been done on the design of parallel algorithms using the SMM. The reader is referred to [3], [4] and the references contained therein.

While one can talk of obtaining the postfix and tree forms for an entire program, we shall limit our disucssion here to simple expressions. These are permitted to contain only operands (constants and simple variables), operators (only the binary operators +, -, *, /, and $\uparrow$ are permitted), and delimiters ('(', and ')').

Like every other parallel algorithm, our algorithms are based on a sequential algorithm. The sequential infix to postfix algorithm we start from is that given by Horowitz and Sahni [6]. This algorithm utilizes a stack as well as a dual priority system. The instack priority (ISP) of an operator or delimiter is the priority associated with the operator or delimiter when it is inside the stack. The incoming priority (ICP) is used when the operator or delimiter is outside the stack. For the operator and delimiter set we are limited to,the priority assignment of Figure 1 is adequate.

The infix to postfix algorithm of [6] is reproduced in Figure 2. This algorithm assumes that the infix expression is in E(1:n) where E(i) is an operator, operand, or delimiter, $1 \leq i \leq n$ (in practice, E(i) will be a pointer into a symbol table). For example, the expression A+B*C is input

as E(1)=A, E(2)=+, E(3)=B, E(4)=*, and E(5)=C. The postfix form is output in P(1:m), m ≤ n. For our example, we shall have P(1)=A, P(2)=B, P(3)=C, P(4)=*, and P(5)=+. The time complexity of procedure POSTFIX is O(n).

operator/delimiter	ISP	ICP
)	-	Ø
↑,unary+,unary-	3	4
*,/	2	2
binary+, -	1	1
(	Ø	4
-oo	Ø	-

Figure 1: Instack and incoming priorities.

```
line procedure POSTFIX(E,P,n,m)
     //Translate the infix expression E(1:n) into postfix//
     //form.  The postfix form is output in P(1:m)//
     //'-oo' is used as bottom of stack character and has//
     //ISP=Ø//
  1  declare n, E(1:n), P(1:m), top, STACK(), i,m
  2  STACK(1) ← '-oo', top ← 1 //initialize STACK//
  3  m ← Ø
  4  for i ← 1 to n do
  5    case
  6      :E(i) is an operand: m ← m+1; P(m) ← E(i);
  7      :E(i)=')':while STACK(top) ≠ '(' do
                     //unstack until '('//
  8                   m ← m+1; P(m) ← STACK(top); top ← top-1
  9                 endwhile
 10                 top ← top-1
 11      :else: while ISP(STACK(top) > ICP(E(i)) do
 12                 m ← m+1; P(m) ← STACK(top); top ← top-1
 13             endwhile
 14             top ← top+1, STACK(top) ← E(i)
 15    endcase
 16  endfor
 17  while top > 1 do //empty stack//
 18      m ← m+1; P(m) ← STACK(top); top ← top-1
 19  endwhile
 20 end POSTFIX
```

Figure 2 Sequential infix to postfix algorithm

While it is often difficult to parallelize algorithms that utilize a stack, in Section 2 we shall see that the algorithm of Figure 2 can in fact be effectivily parallelized. In Section 3, we shall see how the tree form of an infix expression may be obtained in parallel.

2. Parallel Generation of the Postfix Form

Let the infix expression be given in E(1:n) as described in Section 1. We make the added assumption that E does not contain superfluous parenthesis pairs. So, the forms ((A)), (((A))), ((A+B)) are not permitted. Our strategy to determine the postfix form, in parallel, is to determine for each i, a value AFTER(i) such that E(i) comes just after E(AFTER(i)), $1 \leq i \leq n$ in the postfix form. The postfix form of the expression A+B*C is ABC*+. Since E(1:5)=(A,+,B,*,C), AFTER(1:5)=(-,4,1,5,3). Note that B comes just after E(AFTER(3))=E(1)=A; * comes just after E(AFTER(4))=E(5)=C; etc. Since the first token (a token is either an operator or an operand or a delimiter) in postfix form has no predecessor, its AFTER() value is undefined. For convenience, we define AFTER()=Ø for the token that is to come first in the postfix form. So, for the above example, AFTER(1:5)=(Ø,4,1,5,3).

In order to determine AFTER(1:n), we need to first compute the level L(i) of each token in the expression. Informally, the level of a token gives the depth of nesting of parenthesis in which this token is contained. So, if a token is not within any parenthesis, its level is Ø. More formally, the level, L, is defined by the algorithm of Figure 3.

$$\text{step 1: } G(i) \leftarrow \begin{cases} 1 & \text{if } E(i)='(' \\ -1 & \text{if } E(i)=')' \\ \emptyset & \text{otherwise} \end{cases}, \quad 1 \leq i \leq n$$

$$\text{step 2: } L(i) \leftarrow \sum_{j=1}^{i} G(j), \quad 1 \leq i \leq n$$

$$\text{step 3: } L(i) \leftarrow L(i)+1 \quad \text{if } E(i)=')', \quad 1 \leq i \leq n$$

Figure 3 Computation of L.

In Figure 4, we give an example arithmetic expression together with the L() values associated with each token (row 3).

Let us sequence through procedure POSTFIX (Figure 2) as it works on the example expression of Figure 4. When i=1, E(1)='(' and '(' gets put onto the stack. Next, i=2, and E(2)=A is placed into the postfix form. When i=5, the postfix form has P(1:2)=(A,B) and the stack has the form -oo, (, *. During this iteration, * is unstacked (as ISP(*) $\geq$ ICP(E(5))). We shall say that E(3) gets unstacked by E(5).

E(5) gets added to the stack and on the next iteration, E(6)=C is placed in the postfix form. When i=18, the stack has the form -oo, +, ↑, (, -, *, ↑, ↑ and P(1:9)=(A,B,*,C,D,E,F,G,H). During this iteration, E(16)=↑, E(14)=↑, E(12)=*, and E(10)=- get unstacked (in that order). I.e., E(16), E(14), E(12), and E(10) get unstacked by E(18). Furthermore, E(10) is the last operator to get unstacked by E(18).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
E	(	A	*	B	+	C	↑	(	D	-	E	*	F	↑	G	↑	H	)	*	I	-	(	(	J	+	K	)	*	L	)	+	M	)
C	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	-1	0	0	0	1	1	0	0	0	-1	0	0	-1	0	0	-1
L	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	2	3	3	3	3	3	2	2	2	1	1	1
ICP	4		2		1		4	4		1		2		4		4		0	2		1	4	4		1		0	2		0	1		0
ISP	0		2		1		3	0		1		2		3		3		■	2		1	0	0		1		■	2		■	1		■
U			5		21		19			18		18		18		18			21		31				27			30			33		
LU			0		3		0			0		0		0		0		10	7		5				0		25	0		28	21		31
AFTER	■	0	4	2	19	3	10	■	6	12	9	14	11	16	13	17	15	■	20	7	28	■	■	5	26	24	■	29	75	■	32	21	■
Position in P	■	1	3	2	17	4	14	■	5	13	6	12	7	11	8	10	9	■	16	15	23	■	■	18	20	19	■	22	21	■	25	24	■

Figure 4

For each i such that E(i) is an operator, we may define U(i) to be the index in E of the operator or delimiter that causes E(i) to get unstacked. In case E(i) gets unstacked during the while loop of lines 17-19 of procedure POSTFIX, then U(i) = n+1. For our example, U(3) = 5, U(10) = U(12) = U(14) = U(16) = 18. Also, for each i such that E(i) is either an operator or a right parenthesis, we may define LU(i) to be the index of the last operator that gets unstacked by E(i). If no operator is unstacked by E(i), then LU(i) is set to 0. For our example, LU(3)=0, LU(5)=3, LU(7)=LU(10)=LU(12) = LU(14)=LU(16)=0, and LU(18)=10.

Continuing with our example, we see that when i=19, P(1:13)=(A,B,*,C,D,E,F,G,H,↑,↑,*,-), and the stack has the form -oo,+,↑. At this time, E(7)=↑ is unstacked and E(19)=* is stacked. So, LU(19)=7 and U(7)=19. Rows 6 and 7 of Figure 4 give the U, and LU values for all the operators and delimiters of our example. Note that U is defined only for operators and LU only for operators and right parenthesis.

An examination of procedure POSTFIX and our definition of the level L of a token, reveals that if E(i) is an

operator, then:

U(i) = least j, j > i such that ISP(E(i)) $\geq$ ICP(E(j)) and L(i)=L(j). If there is no j satisfying this requirement, then U(i)=n+1.

From the definition of U, it follows that if E(i) is an operator or a right parenthesis, then LU(i) is given by:

LU(i) = least j, j < i such that U(j)=i. If there is no j with U(j)=i, then LU(i)=Ø.

From U and LU, AFTER may be determined as below.

case 1: E(i) is an operand.

In this case, we determine the largest j, j < i such that E(j) is either an operand or LU(j) is defined and greater than Ø (note that as extraneous parenthesis pairs are not permitted, if E(j)=')' then LU(j) > Ø). Such a j does not exist iff E(i) is the first operand in the expression. From procedure POSTFIX and our definition of LU, it follows that

$$\text{AFTER}(i) = \begin{cases} \text{Ø} & \text{if no j as above exists} \\ j & \text{if E(j) is an operand} \\ \text{LU}(j) & \text{otherwise} \end{cases}$$

case 2: E(i) is an operator.

In this case, we see that if there exists a j such that j > i and U(j)=U(i), then AFTER(i) is the smallest j with this property. So, in our example expression, U(1Ø) = U(12) = U(14) = U(16) = 18. Also, in P, E(1Ø) comes immediately after E(12) which comes immediately after E(14). E(14) comes immediately after E(16).

For E(16), however, there is no j, j > 16 and U(j) = U(16). For operators with this property, there are two possibilities: either U(i)-1 is an operand or U(i)-1 is a right parenthesis. If U(i)-1 is an operand, then E(U(i)-1) is the token placed in P just before the unstacking caused by E(i) begins. Hence, AFTER(j) = U(i)-1. If E(U(i)-1) is a right parenthesis, then this right parenthesis would have caused at least one operator to get unstacked (by assumption, extraneous parenthesis pairs are not permitted). Hence, LU(U(i)-1) $\neq$ Ø and E(LU(U(i)-1)) is the operator that immediately precedes E(i) in PE. So, we get:

j $\leftarrow$ least j, j > i and U(j)=U(i)

$$
\text{AFTER}(i) = \begin{cases} U(i)-1 & \text{if } j \text{ is undefined and } E(U(i)-1) \text{ is an operand} \\ LU(U(i)-1) & \text{if } j \text{ is undefined and } E(U(i)-1)=')' \\ j & \text{if } j \text{ is defined} \end{cases}
$$

Row 8 of Figure 4 gives the AFTER values for all the operators and operands in our example expression. The AFTER values link the E(i)s in the order they should appear in the postfix form. This linked list is shown explicitly in Figure 5. From this linked list, we wish to determine the position of each operator and operand in the postfix form. For one of the operands, i.e., the one with AFTER(i)=Ø, this position is already known (it goes into P(1)). With each E(i), let us associate a one bit field K(i). K(i)=Ø iff the position of E(i) in P(i) has not been determined. Initially, K(i)=Ø for all but one of the tokens (i.e. the one with AFTER(i)=Ø).

One may verify that the algorithm of Figure 6, changes all the AFTER values so that on termination E(i) is to occupy position AFTER(i) in P (if E(i) is a delimiter then AFTER(i) is undefined and E(i) does not appear in the postfix form).

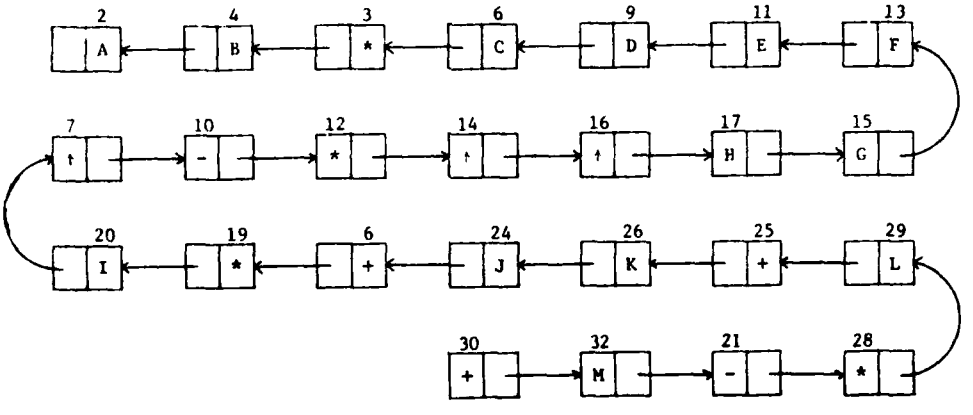


Figure 5

To get a feel for how the algorithm of Figure 6 works, consider the linked list of Figure 7(a). This has 8 nodes in it. AFTER() is shown by an arrow or link. Initially, the first (leftmost) node has K()=1; the remaining nodes have K()=Ø. The K() values are shown outside (and below) the nodes. Node indices are shown outside and above the

```
step 1  //initialize//
        case
           :AFTER(i) is undefined: K(i) ← undefined
           :AFTER(i)=Ø : K(i) ← 1; AFTER(i) ← 1
           :else: K(i) ← Ø
        end case

step 2  //update AFTER//
        for v ← 1 to ⌈ log n ⌉ do
          if K(i)=Ø then  j ← AFTER(i)
                          AFTER(i) ← AFTER(j)
                          if K(j)=1 then K(i) ← 1
                          AFTER(i) ← AFTER(i)+2^(v-1)
                          endif
          endif
        endfor
```

Figure 6 Algorithm to update AFTER

node. In the first iteration of step 2, the linked list splits into two as shown in Figure 7(b) and AFTER(2) is updated to 2. This agrees with the fact that node 2 is the second node (from the left) in Figure 7(a). In the next iteration, each of the two lists of Figure 7(b) split into two and AFTER(4) is set to 3 and AFTER(1) is set to 4. Again, we see that nodes 4 and 1 are respectively the third and fourth nodes in Figure 7(a). In the last iteration the four lists of Figure 7(c) split into 2 each giving the configuration of Figure 7(d). All the AFTER() values now give the position of the respective node in the original linked list.

The correctness of the updating algorithm of Figure 6 may be established formally by providing a proof by induction on the length of the initial linked list. We omit this proof here.

Once the AFTER values have been updated as described above, the postfix form P is obtained by executing the following instruction:

```
if AFTER(i) is defined then P(AFTER(i)) ← E(i)
```

Complexity Analysis

First, let us consider the computation of the levels L (Figure 3). Step 1 can be done in O(1) time using n PEs (each PE is assigned to compute a different G(i)). It can also be done in O(log n) time using n/log n PEs (each PE sequentially computes log n of the G()s). The L(i)s of step 2 may be computed in O(log n) time using n/log n PEs and the partial sums algorithm of [4]. Finally, step 3 can be

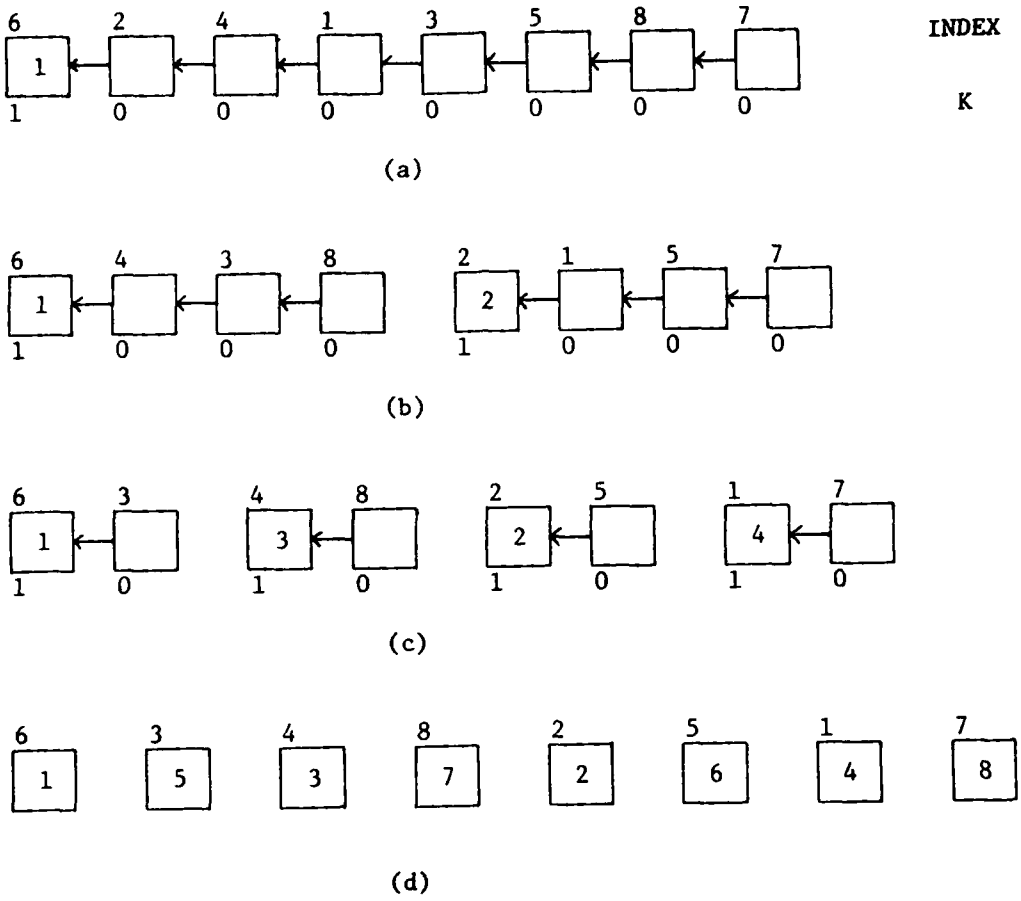


Figure 7

preformed in O(log n) time using n/log n PEs. Hence, the levels L() may be obtained in O(log n) time using n/log n PEs.

Next, consider the computation of U and LU. One possibility is to use mp PEs to first make m copies of each of the p operators and right parenthesis in E (m is the number of operators in E). This takes O(log m) time. Each operator now has a copy of the operators and right parenthesis in E for itself. Each operator E(i) is assigned p PEs to work with. These are first used to eliminate operators and right parenthesis E(j) with $j \leq i$. Next, the level and ISP of E(i) is transmitted to the remaining operators and right

parenthesis. This takes O(log p) time with p PEs. Operators and right parenthesis with a different level number or with ICP > ISP (E(i)) are eliminated. The operators and right parenthesis not yet eliminated are candidates for U(i). The one with least j can be determined in O(log p) time using a binary tree comparison scheme and p PEs. If there are no candidates, U(i)=n+1. LU may now be determined in a similar manner. This strategy to compute U and LU takes $O(n^2)$ PEs and O(log n) time. Using the techniques of [4], it can be made to run in O(log n) time using only $O(n^2/\log n)$ PEs.

An alternative stategy is to first collect togetner all operators and right parenthesis that have the same level number. This can be done in $O(\log^2 n)$ time using n PEs as follows. First, each left parenthesis determines the position of its matching right parenthesis. This is done by simply sorting the left and right parenthesis by their level number. If a stable sort is used, each left parenthesis will be adjacent to its matching right parenthesis following the sort (Figure 8). The sort can be accomplished in $O(\log^2 n)$ time using n PEs [13]. Now, each left parenthesis can determine the address, M(i), of its matcning right parenthesis.

L	1	2		2	2		2	2	3	3	2	1
	(	(		)	(		)	(	(	)	)	)
POSITION	a	b		c	d		e	f	g	h	i	j

(a) before sort

1	1	2	2	2	2	2	2	3	3
(	)	(	)	(	)	(	)	(	)
a	j	b	c	d	e	f	i	g	h

(a) after sort

Figure 8

Once M(i) has been determined for each left parenthesis E(i), we can link together all operators and right parenthesis with the same level as needed in the computation of U. There are only two possibilities for any operator i. These are:

(a) E(i+1)='(': In this case, E(i) is linked to M(i+1)+1.
(b) E(i+1) ≠ '(': In this case i+2=n+1 or E(i+2) is an operator. Regardless, E(i) is linked to i+2.

Performing this linkage operation on the example of Figure 4 gives the linked lists of Figure 9. Now, each linked list can be treated independently. For operators with the highest ISP (i.e., ↑), the U value is obtained by collapsing together consecutive chains of ↑ so that all ↑ point to the nearest non ↑. The U value equals the link value. So, U(7) = 19, U(14) = U(16) = 18. For operators with the next highest ISP, the U values are obtained by removing all nodes representing the operator ↑. The link values give the U value. Doing this on the lists of Figure 9, yields the lists of Figure 10. So, U(3)=5, U(19)=21, U(12)=18,

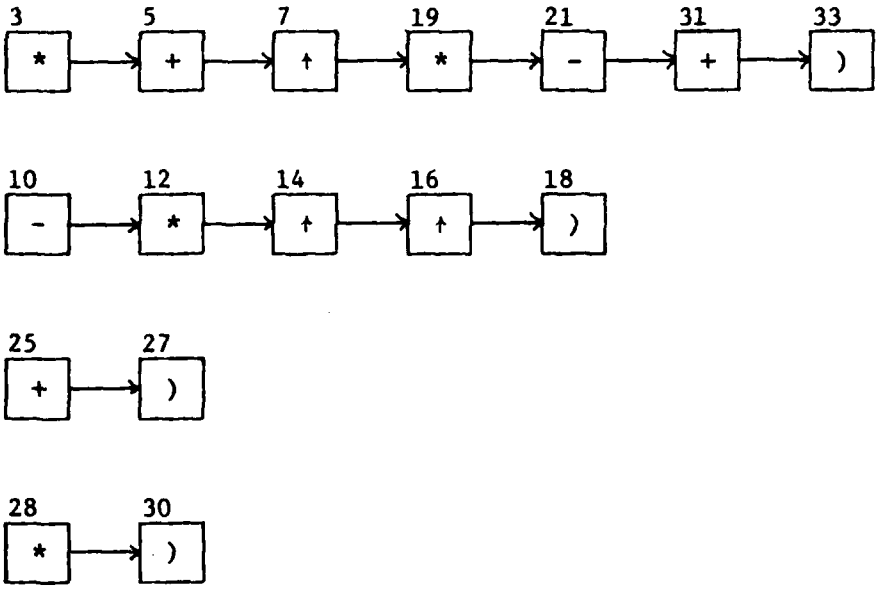


Figure 9

U(28)=30. Now, by eliminating all nodes that represent * and / and collapsing the lists we can determine the U value for the next ISP class. We obtain U(5)=21, U(21)=32, U(10)=18, and U(25)=27. Each elimination and collapsing operation above can be performed in O(log n) time using n PEs and the strategy used in Figure 6 to update AFTER. Since the number of ISP classes is a constant, the time needed to determine U is O(log n).

It should be evident that LU can be computed during the computation of U. Each operator and right parenthesis keeps

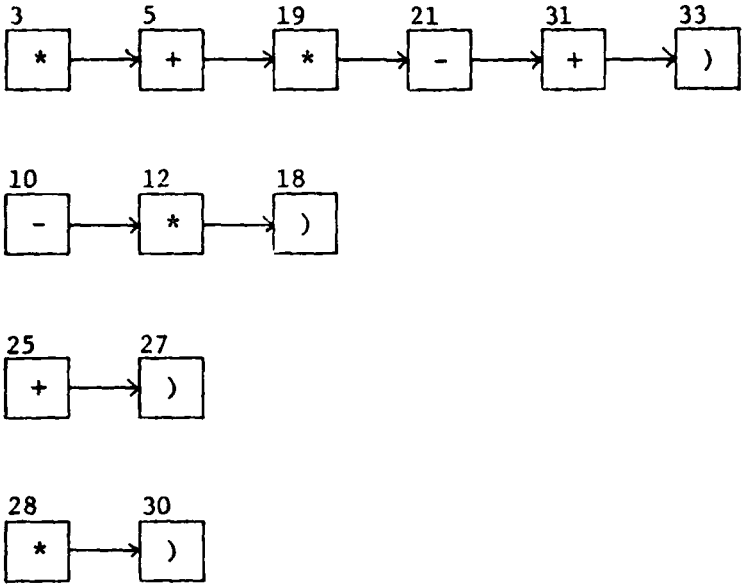


Figure 10

track of the farthest operator it unstacks from each ISP class. The initial values of AFTER() may now be computed. First, each operand determines the nearest (on its left) binary operator, right parenthesis, and operand. These are shown in Figure 11 for our example of Figure 4. Zeroes indicate the absence of a nearest quantity on the left. These three sets of nearest value can be determined in O(log n) time using n PEs. For example, to get the nearest operands, we eliminate all E(i)s that are not an operand. The remaining E(i)s are concentrated to the left. This enables each operand to determine its nearest left operand. Next, the operands are distributed back to their original spots (see [12] for an O(log n) distribution algorithm).

If E(i) is an operand and has no nearest operand on the left, AFTER(i)=Ø. If the nearest binary operator (on the left) has LU() > Ø, then AFTER(i) equals this LU value. If E(i) has a nearest right parenthesis (on the left) then AFTER(i) is the LU value of this parenthesis. Otherwise, AFTER(i) is the location of the nearest operand on the left.

If E(i) is an operator, we can determine the smallest j, j > i such that U(j)=U(i) during the computation of U and LU. So, if such a j exists, AFTER has already been computed. If no such j exists, AFTER(i) is to be set to either U(i)-1 or LU(U(i)-1). Both these quantities are already known. So, the computation of AFTER for operators takes

O(1) additional time.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
E	(	A	*	B	+	C	+	(	D	-	E	*	F	+	G	+	H	)	*	I	-	(	(	J	+	K	)	*	L	)	+	M	)
nearest binary operator		0		3		5			7		10		12		14		16			19				21		25			28			31	
nearest right parenthesis		0		0		0			0		0		0		0		0			18				18		18			27			30	
nearest operand		0		2		4			6		9		11		13		15			17				20		24			26			29	

Figure 11

The updating of AFTER (Figure 6) requires only O(log n) time and n PEs. The formation of P takes O(1) time and n PEs. Hence, using n PEs, the postfix form may be computed in $O(\log^2 n)$ time. The complexity is dominated by the sort step. Another complexity measure worth computing is the EPU (effectiveness of processor utilization). This is:

EPU=

$$\frac{\text{complexity of fastest sequential algorithm}}{\text{complexity of parallel algorithm * no. of PEs}}$$

$$= O\left(\frac{n}{\log^2 n * n}\right)$$

$$= O\left(\frac{1}{\log^2 n}\right)$$

3. Parallel Generation of the Tree Form

As mentioned in Section 1, most code optimizers work on the tree form of an expression. This tree form is easily obtained from the postfix form by considering the algorithm to evaluate postfix expressions. This algorithm is given in Figure 12 (see [6] for an explanation).

Define the degree of an operator to be the number of operands it needs. Let D(i) be the degree of operator P(i). So, D(P(i))=1 if P(i) is unary; D(P(i))=2 if P(i) is a binary operator. Define W(i) as below:

$$W(i) = \begin{cases} 1 & \text{if P(i) is an operand} \\ 1-D(i) & \text{otherwise} \end{cases}$$

```
procedure EVAL(P,n)
   //evaluate the postfix expression P(1:n)//
   declare n, P(1:n),i,STACK
   initialize STACK
   for i ← 1 to n do
      case
       :P(i) is an operand : Put P(i) on the STACK
       :else : remove as many operands from the stack as
               needed to compute P(i). Evaluate P(i) with
               these operands and put the result on the
               STACK
      endcase
   endfor
end EVAL
```

Figure 12

Note that W(i) gives the change in the stack height when procedure EVAL processes P(i) (an operand increases the height by 1 while an operator reduces it by D(i)-1). The stack height, H(i), following the processing of P(i) is given by:

$$H(i) = \sum_{j=1}^{i} W(j) \qquad (3.1)$$

Let us make the simplifying assumption that we have no operator of degree greater than 2.

The tree form of the expression P(1:n) consists of n nodes. Each node has three fields: LCHILD,RCHILD, and P. It is easy to see that LCHILD(i)=RCHILD(i)=Ø for every i such that P(i) is an operand. Also, if P(i) is an operator, then RCHILD(i)=i-1. If P(i) is a unary operator, LCHILD(i)=Ø. This leaves us with the task of determining LCHILD(i) when P(i) is a binary operator. It is not too difficult to see that in this case, LCHILD(i) is the largest j, $j < i$ such that H(j)=H(i).

The LCHILD values for binary operators can therefore be obtained by first computing H(i) as given by (3.1). This can be done in O(log n) time using either n or n/log n PEs and the partial sums algorithm of [4]. Figure 13 shows the postfix form of our example of Figure 4. The W values are given in row 2 and the H values in row 3.

Next the H(i)s are sorted using a stable sort method. This takes $O(\log^2 n)$ time and O(n) PEs [13]. This sort brings a parent and its left child (if the parent is a binary operator) together. So, in our example P(7) and P(9) are brought together. So also are P(6) and P(10); P(5) and P(11); P(4) and P(12); etc. Hence every binary operator can now easily determine its left child. The expression tree

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Prefix	P	A	B	*	C	D	E	F	G	H	↑	↑	*	-	↑	I	*	+	J	K	+	L	*	-	M	+
	W	1	1	-1	1	1	1	1	1	1	-1	-1	-1	-1	-1	1	-1	-1	1	1	-1	1	-1	-1	1	-1
	H	1	2	1	2	3	4	5	6	7	6	5	4	3	2	3	2	1	2	3	2	3	2	1	2	1
	CHILD	0	0	1	0	0	0	0	0	0	8	9	10	11	12	0	14	15	0	0	18	0	20	21	E	23
	CHILD	0	0	0	0	0	0	0	0	0	7	6	5	4	3	0	13	2	0	0	17	0	19	16	E	22

Figure 13

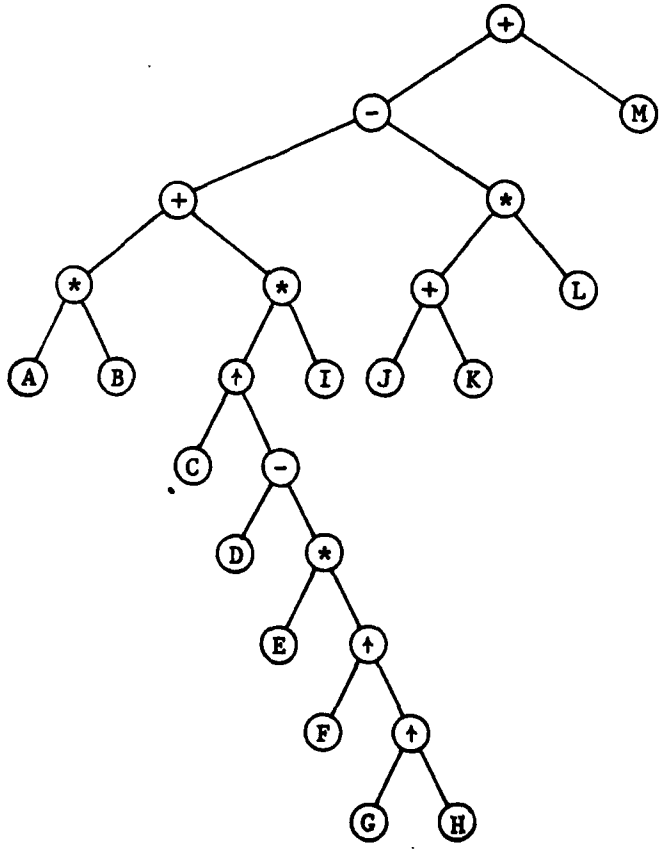


Figure 14

that results for our example is shown in Figure 14.

The additional time needed to obtain the tree is $O(\log n)$ and the number of PEs needed is n. Using the postfix algorithm of Section 2, the tree form may be obtained from the infix form in $O(\log^2 n)$ time using n PEs. The EPU of the resulting tree form algorithm is $O(1/\log^2 n)$.

4. Conclusions

We have shown that it is possible to effectively parallelize the postfix and tree form algorithms. Our parallel algorithms run in $O(\log^2 n)$ time when n PEs are available. If only n/k PEs are available, our algorithms can still be used. The complexity will be $O(k \log^2 n)$.

The results of this paper nicely complement the work reported on the parallel evaluation of expressions (see [1], [2], [8], [10], and [11]).

References

1. Brent, R., "The parallel evaluation of general arithmetic expressions,". J. ACM 21, 2, April 1974 pp. 201-206.
2. Brent, R., Kuck, D.J., and Maruyama, K.M., "The parallel evaluation of arithmetic expressions witnout divisions," IEEE Trans. Comput. C-22, May 1973, pp. 532-534.
3. Dekel, E., and Sahni, S., "Parallel scheduling algorithms," University of Minnesota TR 81-1.
4. Dekel, E., and Sahni, S., "Binary trees and parallel scheduling algorithms," University of Minnesota TR 80-19.
5. Fischer, C.N., "On parsing and compiling arithmetic expressions on vector computers." TOPLS Vol. 2, No. 2, April 1980, pp. 203-224.
6. Horwitz, E. and Sahni, S., "Fundamentals of data structures," Computer Science Press, Patomac, MD, 1976.
7. Knuth, D.E., "An empirical study of FORTRAN programs," Software 1, April 1971, pp. 105-133.
8. Kuck, D.J., "Evaluating arithmetic expressions of n atoms and k divisions in $d(\log_2 n+2 \log_2 k)+ c$ steps, manuscript, March 1973.
9. Kuck, D.J., "Parallelism in ordinary programs," Proc. Symposium on Compleixty of Sequential and Parallel Numerical Algorithms, Carnegie-Mellon, Pittsburgh, PA, May 1973. Academic Press, New York.
10. Kuck, D.J., and Maruyama, K.M., "The parallel evaluation of arithmetic expressions of special forms," Rep. RC4276, IBM Res. Center, Yorktown Heights, NY, March 1973.
11. Maruyama, K.M., "On the parallel evaluation of polynomials," IEEE Trans. Comput.,C-22, Jan. 1973, pp. 2-5.
12. Nassimi, D. and Sahni, S., "Data broadcasting in SIMD computers," IEEE TRANS. on Computers. C-30, no. 2., Feb 1981, pp 101-107.
13. Preparata, F.P., "New parallel-sorting schemes," IEEE Trans. on Computers, C-27, No. 7,July 1978, pp. 669-673.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A116296 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Parallel Generation of Postfix and Tree Forms	5. TYPE OF REPORT & PERIOD COVERED Technical Report April 1981
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Eliezer Dekel and Sartaj Sahni	8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0650
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Minnesota 136 Lind Hall, 207 Church St. SE, Mpls., MN 55455	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, Virginia 22217	12. REPORT DATE April 1981
	13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

A

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Arithmetic expressions, postfix, infix, tree rorm, parallel computing. complexity.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Efficient parallel algorithms to obtain the postfix and tree forms of an infix arithmetic expression are developed. The shared memory model of parallel computing is used.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified


DATE
ILMED
82