Parallel Generation of Postfix and Tree Forms

ELIEZER DEKEL and SARTAJ SAHNI
University of Minnesota Twin Cities

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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*parsing*

General Terms: Algorithms

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

1. INTRODUCTION

The parallel parsing and evaluation of arithmetic expressions has been the focus of research for many years. References [1, 2, 10, 11, 13] are some of the important papers written on the parallel evaluation of arithmetic expressions. The most significant result here is due to Brent [1]. He has shown [1] that arithmetic expressions containing n operands, $n \geq 1$; operators (+, *, and /); and parentheses can be evaluated in $4 \log_2 n + 10(n - 1)/p$ time when p processors are available. Parallel parsing of arithmetic expressions has been considered by Fischer [5], Krohn [8], Lipkie [12], and Schell [16] (among others). Fischer's work is restricted to vector (or pipelined) computers. While Krohn's work was intended primarily for pipelined computers (specifically for the CDC STAR-100), the ideas contained in [8] can be extended to parallel multiprocessor computers. Krohn, however, does not consider the asymptotic performance that could be obtained from his parallel parsing algorithm. Lipkie [12] and Schell [16] explicitly consider parsing on parallel multiprocessor computers. Lipkie [12] provides some grammar rules for parallel parsing but does not develop a formal algorithm. Schell [16] is a thorough study of parallel techniques for several of the phases normally encountered in compiling (scanning, syntax analysis, parsing, error recovery, etc.). Schell develops a parallel LR parser. The complexity of this parser is, however, quadratic

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

Authors' present addresses: E. Dekel, Mathematical Sciences Program, University of Texas at Dallas, Richardson, TX 75080; S. Sahni, Department of Computer Science, University of Minnesota Twin Cities, 136 Lind Hall, 207 Church Street S.E., Minneapolis, MN 55455.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1983 ACM 0164-0925/83/0700-0300 $00.75

in the input size (under some constraints, he shows that its complexity becomes linear). Schell also discusses the applicability of his techniques to precedence grammars.

In this paper, 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 [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, 10, 11, 13] all begin with the tree form of the arithmetic expression.

(4) While the length of individual arithmetic expressions in typical programs is small [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 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, ..., $p - 1$, and an individual PE may be referenced as 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 write simultaneously 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.

Fig. 1. In-stack and incoming priorities.

Operator/Parenthesis	ISP	ICP
)	—	0
$\uparrow$, unary +, unary −	3	4
$*$, /	2	2
binary +, −	1	1
(	0	4
$-\infty$	0	—

```
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). "−∞" is used as bottom of stack character and has ISP = 0.//
  1  declare n, E(1:n), P(1:m), top, STACK( ), i, m
  2  STACK(1) ← '−∞'; top ← 1      //initialize STACK//
  3  m ← 0
  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
```

Fig. 2. Sequential infix-to-postfix algorithm of [6].

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 limit our discussion 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 parentheses ('(' and ')').

The parallel algorithms that we develop here are closely related to the common priority-based sequential infix-to-postfix algorithm. We make explicit reference to the version of this algorithm that is presented in [6]. This algorithm utilizes a stack as well as a dual priority system. The *in-stack priority* (*ISP*) of an operator or parenthesis is the priority associated with the operator or parenthesis when it is *inside* the stack. The *incoming priority* (*ICP*) is used when the operator or parenthesis is outside the stack. For the operator and parenthesis 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,

Table I. Summary of Key Arrays Used in the Parallel Postfix Algorithm

$AFTER(i)$	Index in $E(1:n)$ of token that immediately precedes $E(i)$ in postfix form
$C(i)$	Used to compute $S(1:n)$
$E(i)$	ith token in the infix expression
$G(i)$	Used to compute the depth of nesting of parentheses
$K(i)$	Bit used to designate whether or not the position of $E(i)$ in the postfix form has been determined
$L(i)$	Depth of nesting of parentheses in which token $E(i)$ is contained
$LU(i)$	Index of last operator to be unstacked by $E(i)$ (cf. algorithm of Figure 2)
$P(i)$	ith token in the postfix form of $E(1:n)$
$POS(i)$	Position of $E(i)$ in the postfix form
$S(i)$	Number of tokens in $E(1:i)$ that are not extraneous right parentheses
$U(i)$	Index of token that pops $E(i)$ from the operator stack (cf. Figure 2)

$$\text{Step 1: } G(i) \leftarrow \begin{cases} 1 & \text{if } E(i) = \text{`('}; \\ -1 & \text{if } E(i) = \text{`)'}, \quad 1 \le i \le n; \\ 0 & \text{otherwise.} \end{cases}$$

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

$$\text{Step 3: } L(i) \leftarrow L(i) + 1 \quad \text{if } E(i) = \text{`)'}, \quad 1 \le i \le n.$$

Fig. 3. Computation of L.

or parenthesis, $1 \le i \le n$ (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 $P(1:m)$, $m \le n$. For our example, we have $P(1)$ = A, $P(2)$ = B, $P(3)$ = C, $P(4)$ = *, and $P(5)$ = +. The sequential time complexity of procedure POSTFIX is $O(n)$.

In Section 2, we see that the algorithm of Figure 2 can be effectively parallelized. In Section 3, we 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. For every $E(i)$ that is an operator or an operand, we define a value $AFTER(i)$ such that $E(i)$ comes immediately after $E(AFTER(i))$ in the postfix form of $E(1:n)$. For the first operand in the postfix form, we define the $AFTER$ value to be zero. Note that, since parentheses do not appear in the postfix form, an $AFTER$ value for them need not be defined. As an example, consider the expression (A + B) * C. Its postfix form is AB+C*. Since $E(1:7)$ = ('(', A, +, B, ')', *, C), $AFTER(1:7)$ = (–, 0, 4, 2, –, 7, 3).

Table I provides a convenient summary of all the variables to be used by our parallel algorithm.

Our parallel algorithm to obtain the postfix form of $E(1:n)$ consists of two phases. In the first, the values $AFTER(1:n)$ are computed. In the second phase, the postfix form is obtained using these values. In order to determine $AFTER(1:n)$, first we need to compute the *level* $L(i)$ of each token in the expression. Informally, the level of a token gives the depth of nesting of parentheses in which this token is contained. So, if a token is not within any parentheses, its level is 0. More formally, the level L is defined by the algorithm of Figure 3.

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 $E(2) = \mathrm{A}$ is placed into the postfix form. When $i = 5$, the postfix form has $P(1:2) = (\mathrm{A}, \mathrm{B})$, and the stack has the form $-\infty$, (, $*$. During this iteration, $*$ is unstacked (as $ISP(*) \geq ICP(E(5))$). We say that $E(3)$ gets *unstacked* by $E(5)$. $E(5)$ gets added to the stack, and on the next iteration $E(6) = \mathrm{C}$ is placed in the postfix form. When $i = 18$, the stack has the form $-\infty$, $+$, $\uparrow$, (, $-$, $*$, $\uparrow$, $\uparrow$, and $P(1:9) = (\mathrm{A}, \mathrm{B}, *, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H})$. During this iteration, $E(16) = \uparrow$, $E(14) = \uparrow$, $E(12) = *$, and $E(10) = -$ get unstacked (in that order). That is, $E(16)$, $E(14)$, $E(12)$, and $E(10)$ get unstacked by $E(18)$. Furthermore, $E(10)$ is the last operator to get unstacked by $E(18)$.

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 parenthesis 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 $i = 19$, $P(1:13) = (\mathrm{A}, \mathrm{B}, *, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}, \uparrow, \uparrow, *, -)$, and the stack has the form $-\infty$, $+$, $\uparrow$. At this time, $E(7) = \uparrow$ 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 parentheses of our example. Note that U is defined only for operators and LU only for operators and right parentheses.

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 $LU(i) = 0$.

Before proceeding to determine *AFTER*, it is useful to eliminate extraneous right parentheses. An *extraneous right parenthesis* is formally defined to be one for which the LU value is 0. Extraneous right parentheses together with their matching left parentheses serve no useful function but may be present in E nonetheless. Examples of occurrences of such parentheses are $(\mathrm{A}\underline{)}$, $((\mathrm{A} + \mathrm{B})\underline{)}*\mathrm{C}$, and $(((\mathrm{A} + \mathrm{B})\underline{))}$ (extraneous right parentheses have been underlined).

The elimination of extraneous right parentheses may be accomplished in the following way. Define $C(1:n)$ as below:

$$C(i) = \begin{cases} 0 & \text{if } E(i) = \text{`)'} \text{ and } LU(i) = 0, \\ 1 & \text{otherwise.} \end{cases}$$

	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	)
G	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	25	■	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

Let $S(i)$ be the sum $\sum_{j=1}^{i} C(j)$, $1 \le i \le n$. $S(i)$ gives the number of tokens in $E(1:i)$ that are not extraneous right parentheses. The replacement

$$(E(S(i)),\ U(S(i)),\ LU(S(i))) \leftarrow (E(i),\ U(i),\ LU(i))$$

carried out for all i such that $E(i)$ is not an extraneous right parenthesis results in the elimination of all extraneous right parentheses from E.

As an example, consider the expression

$$(((\mathrm{A} + \mathrm{B} + \mathrm{C})\underline{)} + \mathrm{D}) * (((\mathrm{E})\underline{))}$$

The extraneous right parentheses are underlined. Following the elimination of these parentheses, the expression E takes the form

$$(((\mathrm{A} + \mathrm{B} + \mathrm{C}) + \mathrm{D}) * (((\mathrm{E}$$

As we see below, following the determination of the levels $L(1:n)$, the left parentheses serve no useful function in our algorithm. Hence, they could be eliminated when the extraneous right parentheses are. To accomplish this, we need only define $C(1:n)$ as

$$C(i) = \begin{cases} 0 & \text{if } E(i) = \text{`('} \text{ or } (E(i) = \text{`)'} \text{ and } LU(i) = 0), \\ 1 & \text{otherwise,} \end{cases}$$

and proceed as before.

Once the extraneous right parentheses have been eliminated, *AFTER* may be computed as described below. In the following discussion of the computation of *AFTER*, we assume that n has been updated to the value $S(n)$ defined above.

Case 1. *$E(i)$ Is an Operand.* In this case, we determine the largest j, $j < i$, such that either $E(j)$ is an operand or $LU(j)$ is defined and greater than 0 (note that, as extraneous parenthesis pairs are not permitted, if $E(j) = \text{`)'}$, then $LU(j) > 0$). 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

$$AFTER(i) = \begin{cases} 0 & \text{if no } j \text{ as above exists,} \\ j & \text{if } E(j) \text{ is an operand,} \\ LU(j) & \text{otherwise.} \end{cases}$$

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, $U(10) = U(12) = U(14) = U(16) = 18$. Also, in P, $E(10)$ 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 such that $j > 16$ 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, $AFTER(i) = U(i) - 1$. 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, $LU(U(i) - 1) \ne 0$, and $E(LU(U(i) - 1))$ is the operator that immediately precedes $E(i)$ in P.

So, we get

$$j \leftarrow \text{least } j,\ j > i, \text{ such that } \quad U(j) = U(i);$$

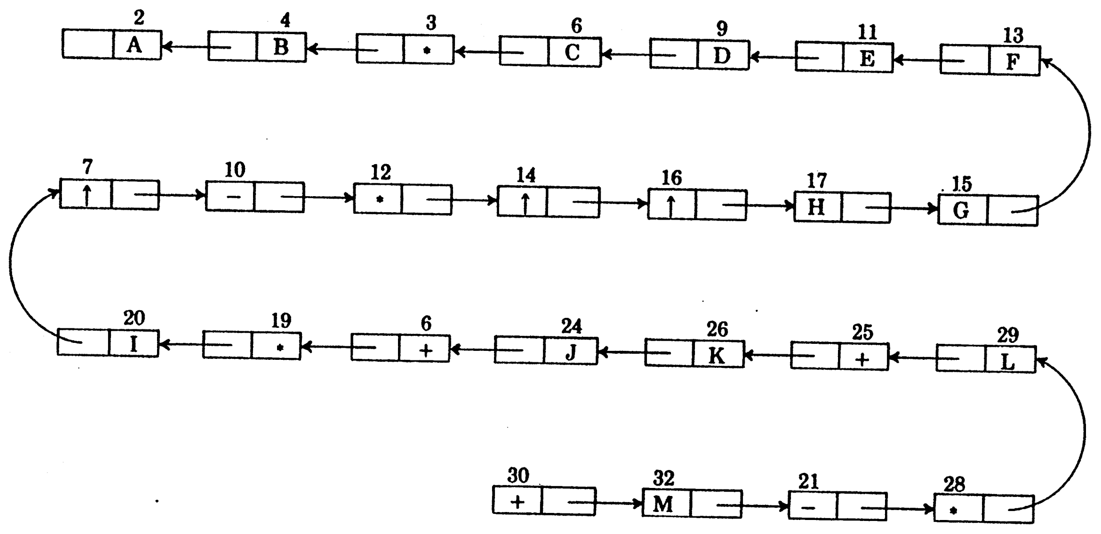
$$AFTER(i) = \begin{cases} U(i) - 1 & \text{if } j \text{ is undefined and } E(U(i) - 1) \text{ is an operand,} \\ LU(U(i) - 1) & \text{if } j \text{ is undefined and } E(U(i) - 1) = \text{`)'}, \\ j & \text{if } j \text{ is defined.} \end{cases}$$

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, *POS*, of each operator and operand in the postfix form. For one of the operands, that is, the one with $AFTER(i) = 0$, this position is already known (it goes into $P(1)$). With each $E(i)$, let us associate a one-bit field $K(i)$. $K(i) = 0$ iff the position of $E(i)$ in $P(i)$ has not been determined. Initially, $K(i) = 0$ for all but one of the tokens (i.e., the one with $AFTER(i) = 0$).

For any node i in the linked list defined by the *AFTER* values, $POS(i)$ is one more than the number of nodes preceding it in that list (the node with *AFTER* value 0 is the first node in the list; so the list is linked backward). The *POS* values may be obtained by recursively splitting this linked list. The first time the list is split, we get two lists (A and B) consisting of alternating elements from the original list. The *POS* value of the first element in list A is already known, and that for the first element of list B is now known to be 2. Figure 6a shows the resulting lists when we start the lists of Figure 5. The lists A and B are split again. When list A of figure 6a is split, we get the lists A1 and A2 of Figure 6b. At this time, the *POS* value, that is, 3, for the first node of list A2 becomes known. Each time a list is split, we get two lists of about half the length. So, following $\lceil \log n \rceil$ splits, all lists will be of size 1, and all the *POS* values will be known. The formal algorithm to determine *POS* is given in Figure 7.

Let us work out a complete example using the algorithm of Figure 7. We start with a smaller list than that of Figure 5. The example list of figure 8 has 8 nodes in it. *AFTER* is shown by an arrow or link. The number inside a node gives its *POS* value (if known). Initially, the first (leftmost) node has $K = 1$; the remaining nodes have $K = 0$. The K values are shown outside and below the nodes. Node indices are shown outside and above the nodes. In the first iteration of step 2, the linked list splits into two as shown in Figure 8b, and $POS(2)$ is updated to 2. This agrees with the fact that node 2 is the second node (from the left) in Figure 8a. In the next iteration, each of the two lists of Figure 8b is split into two, $POS(4)$ is set to 3, and $POS(1)$ is set to 4. Again, we see that nodes 4 and 1 are, respectively, the third and fourth nodes in Figure 8a. In the last iteration the four lists of Figure 8c split into two each, giving the configuration of Figure 8d. All the *POS* values now give the position of the respective node in the original linked list.

The correctness of the algorithm of Figure 7 may be established formally by providing a proof by induction on the length of the initial linked list. We omit that proof here.



2
A
4
B
3
*
6
C
9
D
11
E
13
F
7
↑
10
−
12
*
14
↑
16
↑
17
H
15
G
20
I
19
*
6
+
24
J
26
K
25
+
29
L
30
+
32
M
21
−
28
*

Figure 5

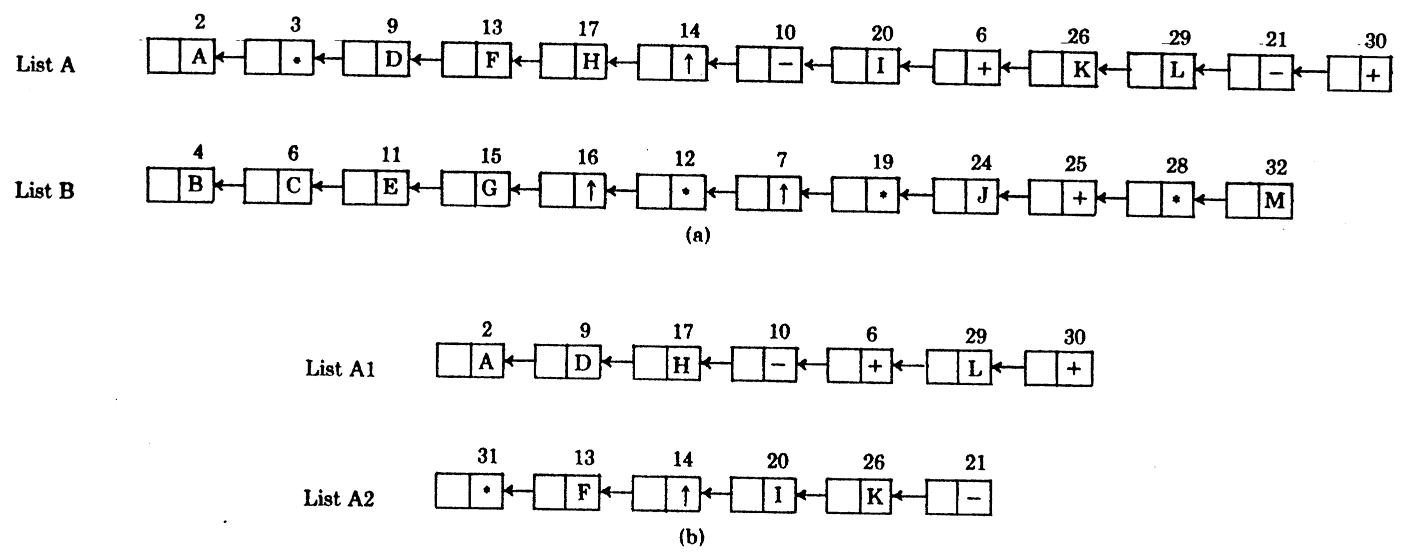


Fig. 6. (a) Splitting the list of Figure 5. (b) Splitting list A of part (a).

Fig. 7. Algorithm to compute *POS*.

```
step 1    //initialize//
  case
    :$AFTER(i)$ is undefined: $K(i) \leftarrow$ undefined
    :$AFTER(i) = 0$: $K(i) \leftarrow 1$; $POS(i) \leftarrow 1$
    :else: $K(i) \leftarrow 0$
  end case
step 2    //split lists and compute POS//
  for $v \leftarrow 1$ to $\lceil \log n \rceil$ do
    if $K(i) = 0$ then $j \leftarrow AFTER(i)$
                       $AFTER(i) \leftarrow AFTER(j)$
                       if $K(j) = 1$ then
                         $K(i) \leftarrow 1$
                         $POS(i) \leftarrow POS(j) + 2^{v-1}$
                       endif
    endif
  endfor
```

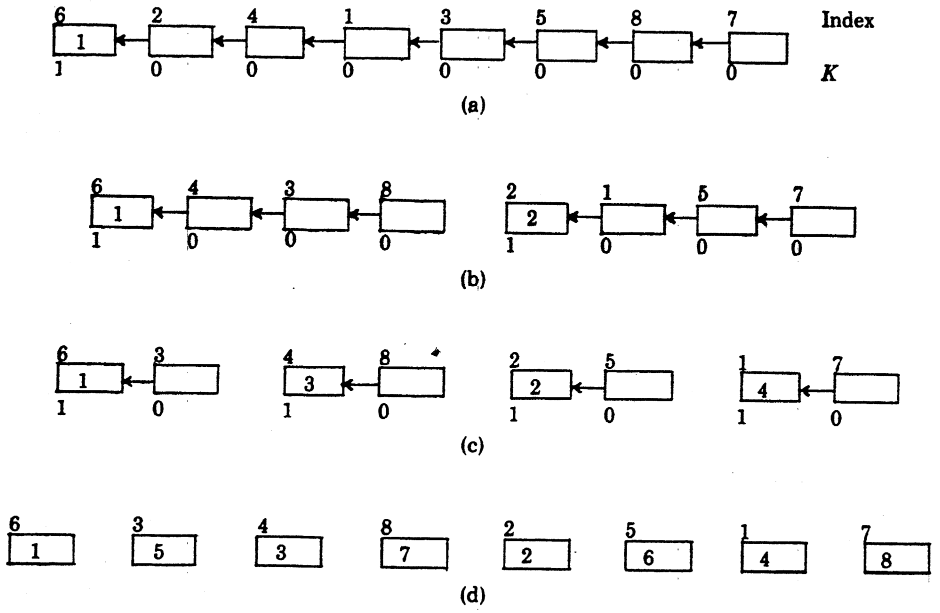


Figure 8

Once the *POS* values have been computed as described above, the postfix form *P* is obtained by executing the following instruction:

if $AFTER(i)$ is defined then $P(POS(i)) \leftarrow E(i)$

2.1 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

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

(a)

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

(b)

Fig. 9. (a) Before sort. (b) After sort.

be performed 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 parentheses in E (m is the number of operators in E). This takes $O(\log m)$ time. Note that $O(\log n)$ time is needed to avoid read conflicts. Each operator now has a copy of the operators and right parentheses in E for itself. Each operator $E(i)$ is assigned p PEs to work with. These are first used to eliminate operators and right parentheses $E(j)$ with $j \leq i$. Next, the level and ISP of $E(i)$ is transmitted to the remaining operators and right parentheses. This takes $O(\log p)$ time (again having no read conflicts) with p PEs. Operators and right parentheses with a different level number or with $ICP > ISP(E(i))$ are eliminated. The operators and right parentheses 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 strategy is to first collect together all operators and right parentheses 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 parentheses by their level numbers. If a stable sort is used, each left parenthesis will be adjacent to its matching right parenthesis following the sort (Figure 9). The sort can be accomplished in $O(\log^2 n)$ time using n PEs [15]. Now, each left parenthesis can determine the address, $M(i)$, of its matching right parenthesis.

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

$E(i + 1) =$ '(': In this case, $E(i)$ is linked to $M(i + 1) + 1$.
$E(i + 1) \neq$ '(': 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 10. Now, each linked list can be treated independently. For

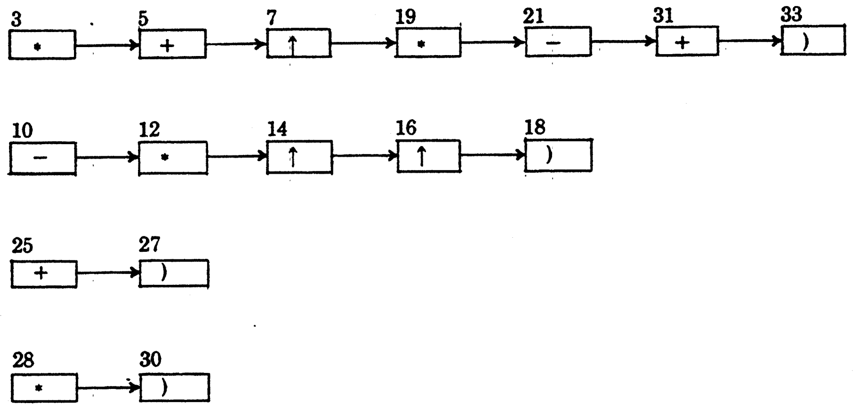


Figure 10

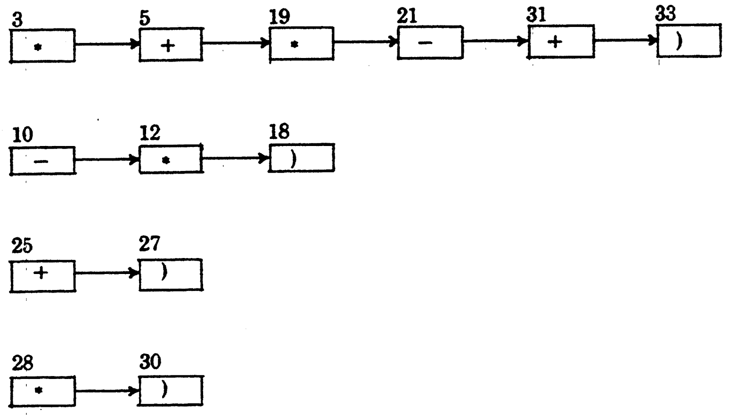


Figure 11

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, $U(7) = 19$, and $U(14) = U(16) = 18$. 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 10 yields the lists of Figure 11. So, $U(3) = 5$, $U(19) = 21$, $U(12) = 18$, and $U(28) = 30$. 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 7 to compute *POS*. 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 track of the farthest operator it unstacks from each *ISP* class. In comparing the two strategies to obtain U and

LU, we note that the first strategy takes $O(\log n)$ time but requires $O(n^2/\log n)$ PEs, while the second strategy takes $O(\log^2 n)$ time and requires only n PEs. So, the log n speedup of the first strategy over the second is obtained through a considerable increase in the number of processors used.

The extraneous right parentheses can be eliminated in $O(\log n)$ time using $(n/\log n)$ PEs. 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 12 for our example of Figure 4. Zeros indicate the absence of a nearest quantity on the left. These three sets of nearest values 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 [14] for an $O(\log n)$ distribution algorithm).

If $E(i)$ is an operand and has no nearest operand on the left, $AFTER(i) = 0$. If the nearest binary operator (on the left) has $LU > 0$, 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.

The computation of *POS* (Figure 7) 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 second strategy to compute U and LU must be used as only n PEs are available). The complexity is dominated by the sort step. Another complexity measure worth computing is the *EPU* (*effectiveness of processor utilization*). This is

$$\begin{aligned} EPU &= \frac{\text{complexity of fastest known sequential algorithm}}{\text{complexity of parallel algorithm} * \text{number of PEs}} \\ &= \Omega\left(\frac{n}{\log^2 n * n}\right) \\ &= \Omega\left(\frac{1}{\log^2 n}\right). \end{aligned}$$

Note also that by using n^2 PEs and the first strategy to compute U and LU, the postfix form can be computed in $O(\log n)$ time. The *EPU* of the resulting algorithm is $\Omega(1/(n \log n))$.

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 13 (see [6] for an explanation).

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 12

Figure 13

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

Prefix	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
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
RCHILD	0	0	2	0	0	0	0	0	0	9	10	11	12	13	0	15	16	0	0	19	0	21	22	0	24
LCHILD	0	0	1	0	0	0	0	0	0	8	7	6	5	4	0	14	3	0	0	18	0	20	17	0	23

Figure 14

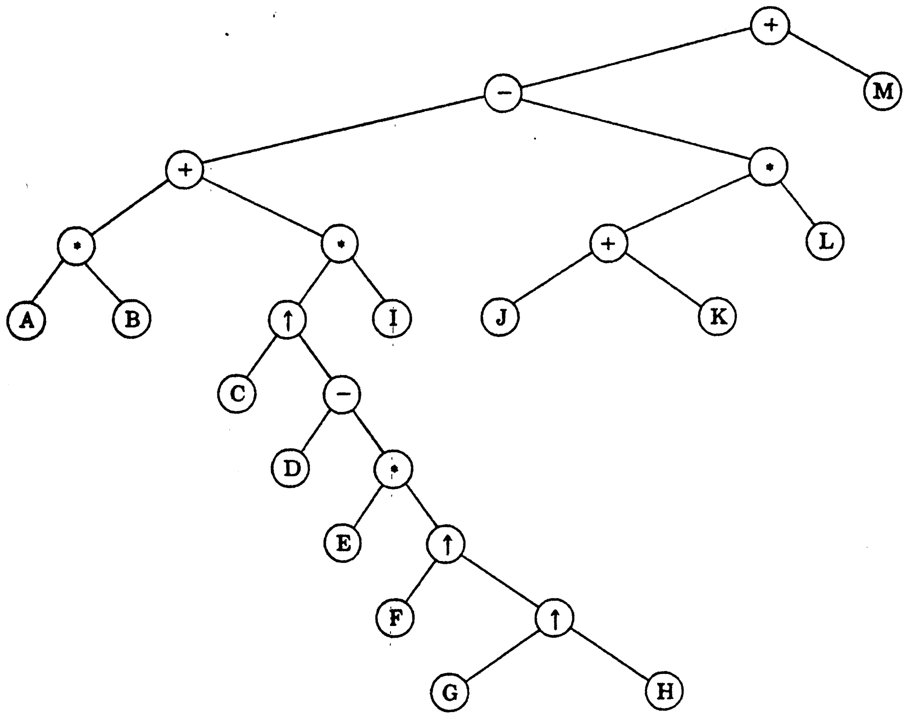


Figure 15

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:

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

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

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

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 $LCHILD(i) = RCHILD(i) = 0$ 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, $LCHILD(i) = 0$. 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 (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 14 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 [15]. 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 that results for our example is shown in Figure 15.

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 $\Omega(1/\log^2 n)$.

4. CONCLUSIONS

We have shown that it is possible to parallelize the postfix and tree form algorithms effectively. 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, 10, 11, 13]).

REFERENCES

1. Brent, R.P. The parallel evaluation of general arithmetic expessions. *J. ACM 21*, 2 (Apr. 1974), 201-206.
2. Brent, R., Kuck, D.J., and Maruyama, K.M. The parallel evaluation of arithmetic expressions without divisions. *IEEE Trans. Comput. C-22* (May 1973), 532-534.
3. Dekel, E., and Sahni, S. Parallel scheduling algorithms. Tech. Rep. TR 81-1, Computer Science Dep., Univ. of Minnesota, Minneapolis, Minn.; *Oper. Res.*, to appear.
4. Dekel, E., and Sahni, S. Binary trees and parallel scheduling algorithms. In *Lecture Notes in Computer Science*, vol. 111: *CONPAR 81*. Springer-Verlag, New York, 1981, pp. 480-492.
5. Fischer, C.N. On parsing and compiling arithmetic expressions on vector computers. *ACM Trans. Program. Lang. Syst. 2*, 2 (Apr. 1980), 203-224.
6. Horowitz, E., and Sahni, S. *Fundamentals of Data Structures*. Computer Science Press, Woodland Hills, Calif., 1976.
7. Knuth, D.E. An empirical study of FORTRAN programs. *Software 1* (Apr. 1971), 105-133.
8. Krohn, H.E. A parallel approach to code generation for Fortran like compilers. In Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines, New York, N.Y., Mar. 18-19, 1975. *SIGPLAN Notices* (ACM) *10*, 3 (Mar. 1975), 146-152.
9. Kuck, D.J. Parallelism in ordinary programs. In *Proceedings, Symposium on Complexity of Sequential and Parallel Numerical Algorithms* (Carnegie-Mellon Univ., Pittsburgh, Pa., May 1973). Academic Press, New York, 1973, pp. 17-47.
10. Kuck, D.J. Evaluating arithmetic expressions of n atoms and k divisions in $(\log_2 n + 2 \log_2 k) + c$ steps. Unpublished manuscript, Mar. 1973.
11. Kuck, D.J., and Maruyama, K.M. The parallel evaluation of arithmetic expressions of special forms. Rep. RC4276, IBM Research Center, Yorktown Heights, N.Y., Mar. 1973.

12. Lipkie, D.E. A Compiler Design for Multiple Independent Processor Computers. Ph.D. dissertation, Computer Science Dep., Univ. of Washington, Seattle, Wash., 1979.
13. Maruyama, K.M. On the parallel evaluation of polynomials. *IEEE Trans. Comput. C-22* (Jan. 1973), 2–5.
14. Nassimi, D., and Sahni, S. Data broadcasting in SIMD computers. *IEEE Trans. Comput. C-30*, 2 (Feb. 1981), 101–107.
15. Preparata, F.P. New parallel-sorting schemes. *IEEE Trans. Comput. C-27*, 7 (July 1973), 669–673.
16. Schell, R.M., Jr. Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment. Ph.D. dissertation, Computer Science Dep., Univ. of Illinois, Urbana, Ill., 1979.

Received April 1981; revised March and August 1982; accepted August 1982