INFORMATION AND COMPUTATION 81, 32–45 (1989)

Optimal Parallel Algorithms for Dynamic Expression Evaluation and Context-Free Recognition

ALAN GIBBONS

Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom

AND

WOJCIECH RYTTER

Institute of Informatics, Warsaw University, Palac Kultury i Nauki 8p, sk.poczt. 1210, 00-901 Warszawa, Poland (present adress) and Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom

We describe a deterministic parallel algorithm to evaluate algebraic expressions in $O(\log n)$ time using $n/\log(n)$ processors on a parallel random access machine without write conflicts (P-RAM) and with no free preprocessing. The input to the algorithm is a string (of the symbols making up the expression) stored in an array. Such a form for the input enables a consecutive numbering of the operands in the expression in $O(\log(n))$ time with $n/\log(n)$ processors. This corresponds to a consecutive numbering of the leaves of the expression tree. This then further permits us to partition the leaves into small segments. We improve the result of Miller and Reif (1985, *in* "26th IEEE Sympos. on Found. of Comput. Sci.," pp. 478–489), who described an optimal parallel randomized algorithm. (Strictly speaking, the input to their algorithm is different being the parse tree of the expression. The input to the innovative part of our algorithm (step 2) is this parse tree which, in addition, has its leaves numbered consecutively from left to right. These two forms are equivalent if we note that such a numbering can be obtained by an optimal parallel algorithm which employs the Euler tour technique and optimal list ranking). Our algorithm can be used to construct optimal parallel algorithms for the recognition of two nontrival subclasses of context-free languages: bracket and input-driven languages. These languages are the most complicated context-free languages known to be recognizable in deterministic logarithmic space. This strengthens the result of Matheyses and Fiduccia (1982 *in* "20th Allerton Conf. on Commun. Control and Comput.") who constructed an almost optimal parallel algorithm for Dyck languages, since Dyck languages are a proper subclass of input-driven languages. Our algorithm includes a new simple method for tree contraction which we call the leaves-cutting method. Its correctness is trival (compared with the method of Miller and Reif) and it can be implemented on a P-RAM without write and without read conflicts.

0890-5401/89 $3.00

1. INTRODUCTION

As a model for parallel computations, we choose the parallel random access machine without write conflicts (P-RAM). The processors are unit-cost RAMs that can access a common memory. Any subset of the processors can simultaneously read the same memory location. However, no two distinct processors can attempt to write simultaneously into the same location. By an optimal parallel algorithm (for a given problem) we mean one that satisfies: $pt = O(n)$, where p is the number of processors used, t is the parallel computation time and where p is close to n (e.g., n, $n/\log(n)$ or $n/\log^2(n)$). Optimal parallel algorithms are known for several simple computational problems. For example, these include computing an associative function of n variables, computing maximum, selection, string matching, and converting an expression to its parse tree.

We take, in the case of algebraic expressions, the same form for the input as that used by Bar-on and Vishkin (1985). The input is a string representing an expression and is stored in an array. Using this string, the operands can be consecutively numbered from left to right. The second step of the algorithm requires that the leaves of the corresponding expression tree should be consecutively numbered from left to right. This is needed to divide leaves into segments. Such a numbering is not easy to make for an arbitrary tree by an optimal parallel algorithm (although it is possible for trees of bounded degree or represented by adjacency lists through the employment of the Euler tour technique and optimal list ranking). However, in our case the leaves (of the tree obtained in step 1) correspond to operands in the expression, and the order of leaves (from left to right) is the same as the order of occurrences of the corresponding operands in the input string. It is easy to consecutively number operands by an optimal parallel algorithm using the 1-dimensional array representing the expression. We think that this form for the input, in the case of expressions, is the most natural one.

Dynamic expression evaluation was defined by Miller and Reif (1985) as the problem of evaluating an expression with no free preprocessing. Miller and Reif (1985) gave a deterministic almost optimal parallel algorithm for this problem. Independently, a similar algorithm was described by Rytter (1985a) for the computation of recursive programs with independent calls (expressions are a special case of such programs). Bar-on and Vishkin (1985) constructed an optimal parallel algorithm (with $t = O(\log n)$ and $p = n/(\log n)$) to convert an expression to its parse tree. Here we use their result along with a method which we call leaves-cutting. An invariant of the leaves-cutting operation is that each internal node of the trees has two sons. Hence a reduction in the number of leaves also implies a proportional reduction in the total number of nodes. For the reduced tree we could use

the almost optimal parallel algorithms of Miller and Reif or of Rytter. However, we use the same approach as that used in the preprocessing phase.

Many efficient parallel algorithms are based on the following processing of a binary tree of $\log(n)$ height in bottom-up fashion (the root is located at the top). We consider a pair of leaves with the same father. The value of the father is computed and the two leaves are cut (removed). Their father becomes a new leaf with a computed value. Such an operation can be applied to every pair of leaves with the same father in parallel because these cuttings are "independent". After $\log(n)$ parallel steps of this type, the value of the root is computed and the root is a leaf. In the leaves-cutting technique we generalize this to arbitrary trees which may be of height $O(n)$. Two main concepts are first the operation of cutting a single leaf and second the "independence" of such operations. These are defined in step 2 of our algorithm for expression evaluation. The key property of the operation of cutting is that a leaf can be cut even if its brother is not a leaf. A tree which results after applying in parallel many independent cuttings is again a binary tree because each internal node has two sons. In the tree contraction of Miller an Reif we can have a tree with possibly many long chains. The main advantage of our leaves-cutting method (compared with that of Miller and Reif or of Rytter) is that verification of the upper bound of $O(\log(n))$ on the number of iterations becomes trivial. Moreover, the method can be easily implemented without read conflicts.

Bracket languages are context-free languages. Any sentence of a bracket language is a string with an explicitly encoded parse tree. It is generated by a grammar whose productions are of the form $A \to (u)$, where u is a string not containing brackets. The strings generated by such grammars are generalizations of bracketed algebraic expressions.

The recognition problem for bracket languages can be easily transformed into the problem of evaluating certain algebraic expressions. Operations in these expressions act on sets of nonterminals and have a syntactic character. The optimal parallel algorithm for dynamic expression evaluation implies an optimal parallel algorithm for the recognition of such languages. Matheyses and Fiduccia (1982) gave an optimal parallel algorithm for the recognition of Dyck languages. Dyck languages are a proper subclass of input driven languages (i.d. languages). An almost optimal parallel algorithm for the recognition of these languages was given by Giancarlo and Rytter (1985). I.d. languages are accepted by very restricted one-way deterministic pushdown automata (dpda's). The reading of some input symbols is defined to push or to pop (stack) symbols on the pushdown store. In the case of Dyck languages opening brackets increase the height of the stack by one, and closing brackets cause a similar decrease. The corresponding dpda can be easily constructed to accept well-composed

strings of brackets perhaps containing many types of brackets (e.g., [,] and (,)).

The recognition problem for i.d. languages can again be reduced to the computation of certain expressions whose algebraic operations correspond to the program of the corresponding dpda. The biggest subclass of context-free languages which is know to be recognizable in $O(\log(n))$ time is the class of unambiguous cfl's. However, in this case we do not anticipate an optimal parallel algorithm. The known construction uses n^7 processors, (Rytter, 1987). Rytter (1985c) has shown that general context-free recognition can be done on perfect-shuffle and on cube-connected computers in $\log^2 n$ time using n^6 processors. It is an interesting question whether our optimal parallel algorithms for bracket and i.d. languages can be simulated on a perfect shuffle or on a cube-connected computer without substantially increasing the complexity.

2. An Optimal Parallel Algorithm for Dynamic Expression Evaluation

In this section we describe the algorithm for dynamic expression evaluation. We concentrate upon the evaluation of arithmetic expressions in which we allow the operations of addition, subtraction, multiplication, and division. We assume that their cost is $O(1)$. As we point out later, our algorithm also works for any fixed algebra with a finite carrier (set of elements). This includes syntactic algebras corresponding to context-free grammars (where the carriers are sets of nonterminals) and algebras corresponding to dpda's.

The main result of the section is the following:

THEOREM 1. *Given the expression as a strong stored in array, then*

(a) *arithmetic expressions of size n can be computed on a P-RAM in* $O(\log(n))$ *time using* $n/\log(n)$ *processors.*

(b) *algebraic expressions of size n with operations from an algebra with carrier of* $O(1)$ *size can be computed in* $O(\log(n))$ *time using* $n/\log(n)$ *processors.*

Proof. We present an algorithm which has the stated properties of the theorem. Initially, our description provides an algorithm which runs in $O(\log(n))$ time using $n/2$ processors. A straightforward application of Brent's theorem, which we shall explicitly describe, then easily brings about a reduction in the number of processors to $n/\log(n)$ without detriment to the parallel running time. For ease of exposition, we assume that n is a power of 2 and that n is divisible by $\log(n)$. This is true for our example of Fig. 3.

ALGORITHM. The input to the algorithm is the string of the expression symbols stored in an array.

Step 1.

The output from this step will be the binary tree representation of the expression and, in addition to which, the leaves of the tree (where the operands are stored) will be consecutively numbered from left to right. The optimal parallel algorithm of Bar-On and Vishkin (1985) can be employed to obtain the binary tree representation. The additional requirement that the leaves be numbered from left to right is obtained as follows. First, the operands within the initial array representation are numbered from left to right as follows. We mark each element in the array if and only if it is an operand. The problem is then reduced to the problem of consecutively numbering marked elements. We assign the number 1 to each marked element and the number 0 to all other elements. The required rank, within the array, of each operand can then be obtained by a parallel prefix computation in $O(\log(n))$ time using $n/\log(n)$ processors (for example, see Kindervater and Lenstra, 1985). Now the consecutive numbering of the leaves of the tree is obtained simply by carrying the rank of each operand along with the operand throughout the employment of the algorithm of Bar-On and Vishkin.

end of Step 1.

Central to step 2 of the algorithm will be the operation we call **leaves cutting.** As we shall define more precisely, one operation of leaves cutting consists of reducing the size of the expression tree by the parallel removal of some leaves of the tree. We introduce the operation by first describing how a single leaf may be removed by a local reconstruction of the tree.

A functional form (an algebraic expression) $f_i(x)$ is associated with each internal vertex v_i. Initially $f_i(x) = x$. When computing the value associated with v_i the functional form $f_i(x)$ is used in the following way. If $\diamond$ is the operator associated with v_i and the sons of v_i have the associated values $c1$ and $c2$, then the value of v_i is given by substituting $(c1 \diamond c2)$ for x in $f_i(x)$. With reference to Fig. 1, consider now removing the single leaf v_2. Imagine (for the moment) that no other leaves have been removed. Let v_2 have the associated constant value c. If v_1 is not a leaf and father(v_2) is not the root of the tree, then the tree is reconstructed locally as indicated within Fig. 1. The value of the subtree rooted at v_1 is represented by x, so that after reconstruction, the vertex v_1 requires a means of storing the functional form $(x \diamond c)$. Here $\diamond$ is the operator associated with v_3. The value of the function now stored at v_1 is of course the value of the subtree originally

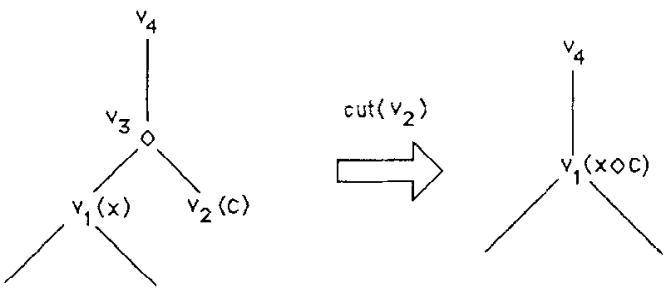

FIGURE 1

rooted at v_3 and so the new tree computes the same expression as the original tree. If father(v_2) is the root of the tree, then in these examples, v_4 is removed from Fig. 1.

Thus we have described the cutting of a first and just one leaf. If x represents the value of the subtree rooted at v_i in the original tree, then in general we wish to store at v_i the functional form $f_i(x)$ resulting from possibly many such reductions of the expression tree. After the single first reduction just described, $f_i(x)$ is set to the expression $(x \diamond c)$. However, we need to describe the general removal of a reducible leaf after the tree has already been subjected to many reductions. We do this schematically according to Fig. 2. Here again, without loss of generality, father(v_2) is presumed not to be the root of the tree and v_1 is assumed not to be a leaf. If v_1 is a leaf then instead of storing a functional form at v_1 we store a value which results from evaluating (in this case) a subtree with leaves v_1 and v_2.

A crucial observation concerning the cutting of a single leaf is that it can always be achieved in constant bounded time. Concider the respecification of the functional form $f_1(x)$. Given the set of operators $\{+, -, *, /\}$ there is a simple inductive argument that the most general form that any $f(x)$ can attain is:

$$f(x) = ((ax + b)/(dx + e)).$$

By a suitable choice of the constant coefficients a, b, d, and e we can represent any specific functional form that may arise. Thus we need only store the values of these coefficients in order to represent the expression $f(x)$. Initially, and for all internal vertices, $a = e = 1$ and $b = d = 0$. For each reduction thereafter the coefficients are easily recomputed in constant bounded time. This is because (as described in Fig. 2) respecification of $f_1(x)$ merely involves a composition of two functions, namely $f_3(f_1(x) \diamond c)$, each of which is restricted by the general form just described. Having respecified $f(x)$, local reconstruction of the tree is easily achieved in $O(1)$ time by the movement of father–son pointers.

Now consider the cutting of several leaves in parallel. Let cut(v_2) be the operation of removing leaf v_2 and respecifying the functional form

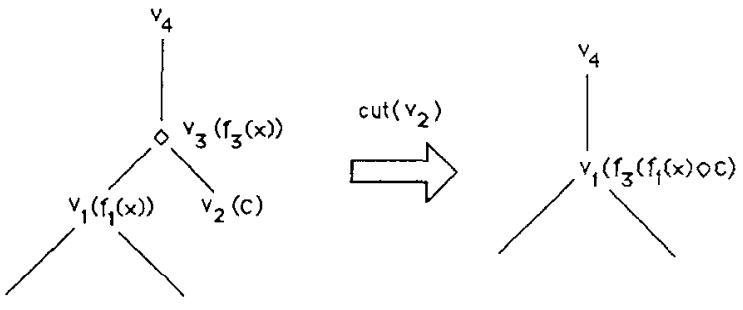

FIGURE 2

associated with v_1. Here we again refer to Fig. 2. In this operation v_1 becomes the son of v_4. We define:

$$\text{involved}(v_2) = \{v_1, v_2, v_3\}.$$

We say that two operations cut(v) and cut(v') are independent if and only if

$$\text{involved}(v) \cap \text{involved}(v') = \varnothing.$$

A crucial property of the leaf cutting operation is that a set of mutually independent operations can be performed in parallel without read and without write conflicts. By the type of a leaf, we mean that it either a left or a right son of an internal vertex. A sufficient condition that the operations cut(v) and cut(v') are independent is that the two leaves v and v' are non-consecutive and that they are of the same type.

For the purpose of the second (and final) step of the algorithm the operation of **leaves-cutting** is defined as follows. First we associate a processor with each odd numbered leaf. Then for all such leaves v which are left sons we perform in parallel the operation cut(v). Next for all such leaves v which are right sons we perform in parallel cut(v). In this way, the number of remaining leaves does not exceed $n/2$. For a second application of the leaves-cutting operation we will need a consecutive numbering of these remaining leaves from left to right (to associate processors with operations cut(v) in $O(1)$ time). However, for each such leaf its new number is easily computed on an EREW P-RAM in $O(1)$ time by halving its old number. After one application of the leaves-cutting operation the number of leaves is reduced by a factor $\frac{1}{2}$. Hence a logarithmic number of consecutive applications of leaves-cutting is sufficient to reduce the expression tree to a single node and so evaluate the arithmetic expression. Thus we summarise the second step of the algorithm (in which there are no read conflicts (nor write conflicts)) as follows:

Step 2.
for $i = 1$ **to** $\lceil \log_2 n \rceil$ **do** perform the operation of **leaves-cutting**
end of Step 2 (and of the algorithm).

The algorithm as we have described it runs in $O(\log n)$ time using $n/2$ processor, where n is the number of operands in the arithmetic expression input to the algorithm. Suppose, however, that we have p processors, where $1 < p < n/2$. Within the algorithm there is now an initial sequence of leaves-cutting operations (about $\log(n/p)$ in number) in each of which there are insufficient processors to assign one to each odd-numbered leaf. Just prior to the $(i+1)$th such leaves-cutting operation the number of leaves of the tree is $n/2^i$. We (in effect) partition these leaves into segments, from left to

right, the first $(p-1)$ segments each contain $\lceil n/p2^i \rceil$ leaves whilst the final segment contains $(n/2^i-(p-1)\lceil n/p2^i \rceil)$. Each of the available processors is now uniquely assigned to one of these segments of leaves (this is possible in constant time since we have a left to right numbering of the leaves) and (in parallel with the other processors working on their own segments) performs the leaves-cutting operation in sequential style for that segment. The number of sequential cutting steps (in each of which there is parallel cutting of leaves, at most one leaf from each of the current segments) in this period of processor multiplexing is then $(n/p)(1+1/2+1/4+1/8\cdots 1/2^k)$, where $n/p \approx 2^{k+1}$. Thus, $O(n/p)$ time is sufficient for this phase. The subsequent and final sequence of leaves-cutting operations (in which for each leaves-cutting operation there are sufficient processors to assign one to each odd-numbered leaf) takes a furthur $O(\log p)$ time to evaluate the expression.

If in the considerations of the previous paragraph we let $p = n/\log n$, then the proof of part (a) of the theorem follows. When the input expression is an algebraic expression with operations from an algebra with a constant number of elements a slight modification of the algorithm is required. Otherwise the proof of part (b) of the algorithm follows as part (a). Instead of functional forms (being associated with the internal nodes of the tree) we use functions, whose values and arguments are elements of the algebra. These functions will reflect the dependence of the value associated with one node on the value associated with another node in the leaves-cutting operation. The description of these functions is of constant size, since the number of values and arguments is constant. Essentially the whole algorithm works in the same way. The time complexity and the number of processors used are not affected. This completes the proof of the theorem. ∎

We illustrate the algorithm through the evaluation of the arithmetic expression whose parse tree (as output from step (1) is shown in Fig. 3(a). Here the order (from left to right) of each leaf is shown and for each leaf the bracketed number is the value stored at that leaf. The particular expression evaluated here uses the operations of addition and multiplication only. This in turn means that the most general form that the functional forms stored at each internal node can attain is $(ax+b)$. Initially for each node $a=1$ and $b=0$. Subsequent respecification is indicated in brackets alongside the appropriate internal nodes. In the same figure, (b) illustrates one leaves-cutting operation. On the left is the tree after all odd numbered leaves which are left sons have been cut. On the right is the same tree after the odd numbered nodes which are are right sons have now been cut. The remaining leaves are now all even numbered and the final step of the leaves-cutting operation is to halve these numbers (in constant time)

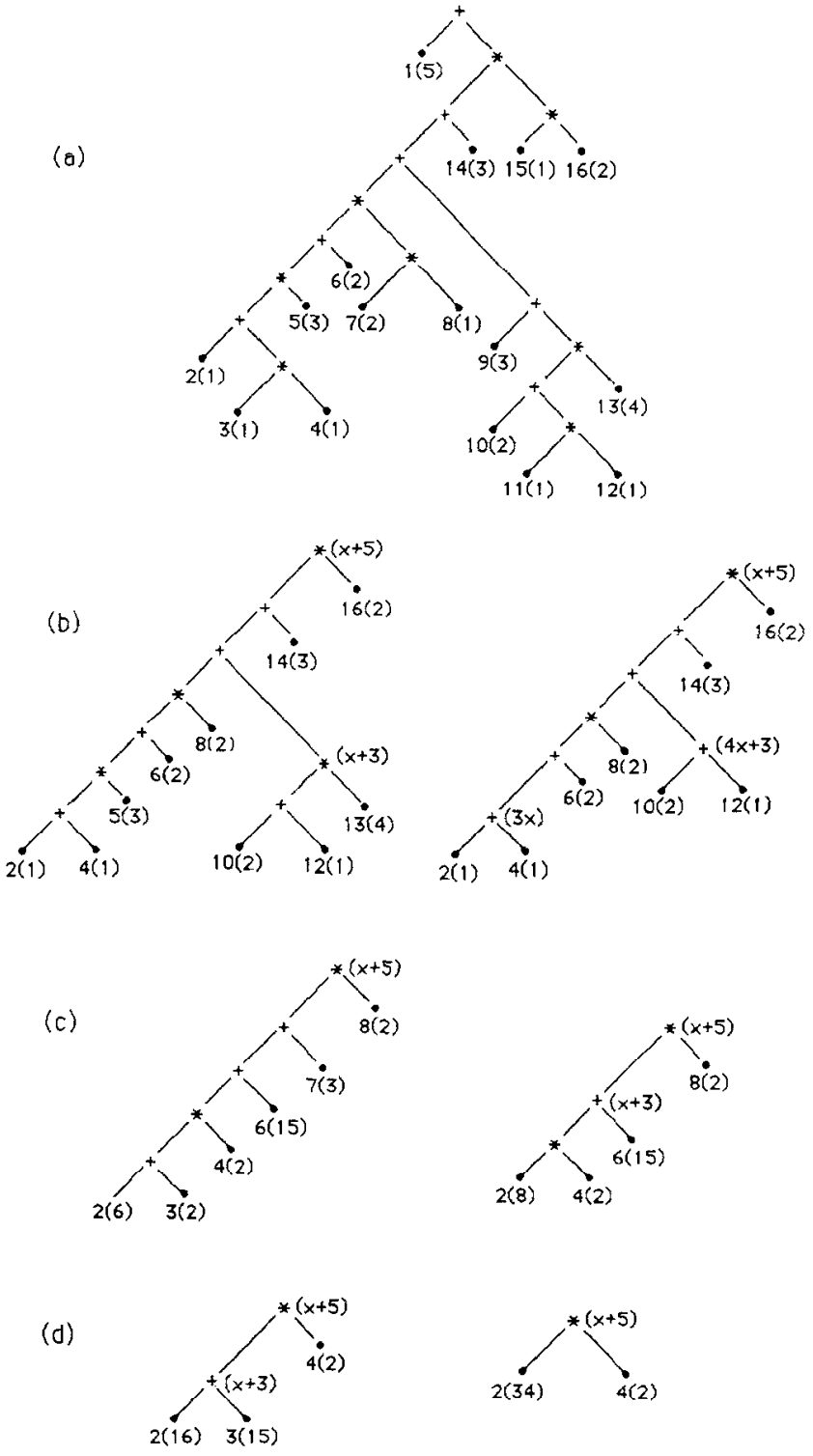
(a)
1(5)
14(3) 15(1) 16(2)
6(2)
5(3) 7(2) 8(1)
2(1)
3(1) 4(1)
9(3)
13(4)
10(2)
11(1) 12(1)
(b)
*(x+5)
16(2)
14(3)
8(2)
6(2)
5(3)
2(1) 4(1)
(x+3)
13(4)
10(2) 12(1)
*(x+5)
16(2)
14(3)
8(2)
6(2)
(3x)
2(1) 4(1)
(4x+3)
10(2) 12(1)
(c)
*(x+5)
8(2)
7(3)
6(15)
4(2)
2(6) 3(2)
*(x+5)
8(2)
(x+3)
6(15)
2(8) 4(2)
(d)
*(x+5)
4(2)
(x+3)
2(16) 3(15)
*(x+5)
2(34) 4(2)

FIGURE 3

before the next application of the leaves-cutting operation. This is illustrated in similar fashion in (c), then (d) shows the subsequent operation. One more application of leaves-cutting now yields the value $((34 * 2) + 5) = 73$ for our example expression.

3. The Parallel Recognition of Bracket and Input-Driven Context-Free Languages

A context-free grammar is given by a 4-tuple $G = (N, T, P, S)$, where N is the set of nonterminals, T is the set of terminal symbols, P is the set of productions, and S is a starting nonterminal symbol. G is a bracket grammar if and only if each production is of the form $A \to (u)$, where u does not contain the brackets "(" and ")." For ease of exposition we shall assume that G is in a Chomsky-like normal form. Each production is then of the type $A \to (BC)$ or of the type $A \to (a)$, where B and C are nonterminals and a is a terminal.

By a recognition problem we mean a problem of the following type. Given an input string w and a grammar G, decide whether w is generated by G. The size of the problem is the lenght n of the input string w. As we now see, any specific recognition problem can be reduced to the evaluation of a certain algebraic expression. We define the operation $*$ on sets of nonterminals as follows:

$$S1 * S2 = \{A \mid A \to BC \text{ is a production and } B \in S1,\ C \in S2\}.$$

Consider the example with productions

$$S \to (CB)\ S \to (AS)\ A \to (BB)\ A \to (a)\ B \to (AC)\ B \to (b)\ B \to (a)\ C \to (BB)$$

and input string

$$w = ((a)(((b)(b))(a))).$$

For every terminal symbol x replace each substring "(x)" in w by a set $\{A \mid A \to (x)\}$. The string obtained contains brackets and sets of nonterminals as symbols. Next in this string for every two adjacent sets of nonterminals (in the substring of the form (XY) insert the operation $*$ between X and Y (obtaining the substring $(X * Y)$). Finally, for every set X of nonterminals surrounded by brackets (not adjacent to any other set Y) insert the operation $*$ between X and an adjacent bracket as follows: replace $(X($ by $(X * ($ and replace $)X)$ by $) * X)$. For the example string we then obtain

$$(\{A, B\} * ((\{B\} * \{B\}) * \{A, B\})).$$

The string so obtained is an algebraic expression. In the example, the value of the expression is $\{S\}$. It can be easily seen that the input string w is generated by a grammar if and only if the value of the corresponding expression is a set containing S. Hence the problem of recognition is reduced to the computation of an algebraic expression. The underlying algebra has a carrier of $O(1)$ size. Hence Theorem 1 gives:

THEOREM 2. *The recognition problem for bracket languages can be solved on a P-RAM in* $\log(n)$ *time using* $n/\log(n)$ *processors.*

A context-free language L is said to be input-driven if and only if there is a one-way deterministic pushdown automaton A accepting L such that in each move the change of the stack height of A is determined by the scanned input symbol alone. We say that the automaton A is normalized if and only if it changes the height of its stack by one when scanning one input symbol.

Without loss of generality, we first show that A can always be presumed to be normalized. If a is an input symbol which causes $k > 0$ symbols to be pushed then we can replace a by the string $h(a) = a_1 a_2 \cdots a_k$. Here each a_i causes one symbol to be pushed and $h(a)$ makes the same change to the stack as the symbol a alone. If symbol a does not change the height of the stack then we can replace it by $h(a) = a_1 a_2$ such that a_1 causes a push move and a_2 causes a pop move. In this way the automaton A, for the input x, is simulated by some normalized automaton A' on the input $h(x)$. Instead of checking if A accepts x we check if A' accepts $h(x)$. The string $h(x)$ has length $O(n)$, and can be constructed easily from x by an optimal parallel algorithm. Hence we can assume that any pushdown automaton A accepting an input-driven language is normalized. We can also assume, without loss of generality, that when A accepts then the height of the stack is one. We simply add a sequence of extra symbols if necessary.

THEOREM 3. *The recognition problem for i.d. languages can be solved on P-RAM in* $\log(n)$ *time using* $n/\log(n)$ *processors.*

Proof. We can assume that the pushdown automaton accepting an input-driven language L is normalized. Now push symbols correspond to opening brackets and pop symbols correspond to closing brackets. These brackets give the structure of a certain expression. The recognition problem can be again reduced to the problem of computing an algebraic expression. Let $P(M)$ be the set of all subsets of the set $M = (S \times W) \times (S \times W)$, where $\times$ denotes the Cartesian product of sets. S is the set of states and W is the pushdown alphabet of the automaton A. $P(M)$ is the set of binary relations on $(S \times W)$. Such relations are values corresponding to certain substrings

of the input string. The value corresponding to a substring v of the input string is the set:

$\{((s1, q1), (s2, q2)) \mid$ automaton A starting in the state $s1$ with $q1$ as the only element of its stack after reading v is in the state $s2$ with $q2$ as the only element of the stack$\}$.

The operations on such sets (relations) can be made to reflect the behaviour of the automaton. Such an approach was used earlier by Rytter (1986) to design a space-efficient algorithm. We specify now in more detail the relationship between input words and expressions. The input word will be transformed into an expression in a way analogous to that used in the case of bracket languages.

We introduce two operations $*$ and &. The first operation is the composition of relations (elements of $P(M)$). The second operation is rather technical. Let Q be the set of all pairs $[a, b]$ of symbols of the input alphabet such that a is a push symbol and b is a pop symbol. For each such pair $[a, b]$ and relation $R \in P(M)$ we define:

$[a, b]$ & $R = \{((s1, q1), (s2, q2)) \mid$ A starting in state $s1$ with $q1$ as the only element on the stack, after reading a pushes a symbol $q1'$ on the stack and changes the state to $s1'$ and A starting in the state $s2'$ with top element $q2'$ after reading the symbol b changes the state to $s2$, for some $((s1', q1'), (s2', q2')) \in R$. Moreover, $q1 = q2\}$.

Informally $[a, b]$ & R is the set of all pairs which are "below" pairs from R in the "context" $[a, b]$.

For an input word x we construct an expression $\mathrm{ex}(x)$ involving the operations $*$ and &. We illustrate the construction on an example.

Let $x = a_1 a_2 \cdots a_{10}$ be the input word for which the history of the computation of the automaton A is indicated in Fig. 4. We can see from the figure that the symbols a_1, a_2, a_4, a_5, a_8 are push symbols and the symbols a_3, a_7, a_9, a_{10} are pop symbols.

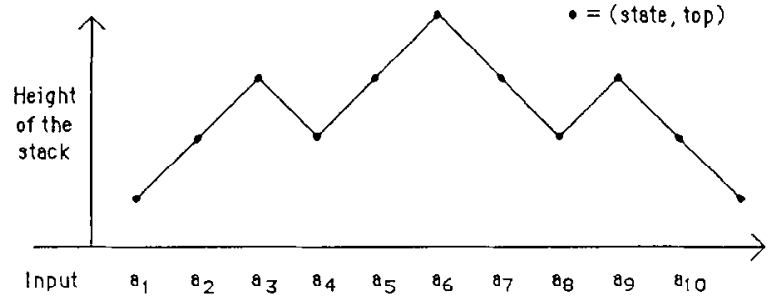

FIGURE 4

First we insert the bracket "(" immediately before every push symbol and the bracket ")" immediately after every pop symbol. In this way the following string $x1$ is obtained:

$$(a_1(a_2a_3)(a_4(a_5a_6)a_7)(a_8a_9)a_{10}).$$

Next for every substring of the form (ab) we compute the relation

$$R(a, b) = \{(s1, q1), (s2, q2)) \mid \text{A starting in state } s1 \text{ with the stack containing only } q1 \text{ after reading } ab \text{ can be in state } s2 \text{ with the stack containing only } q2\}.$$

Every substring of the form (ab) is replaced by $(R(a, b))$. Suppose that $R(a_2a_3) = R_1$, $R(a_5, a_6) = R_2$ and $R(a_8, a_9) = R_3$. Then string $x1$ is transformed into

$$x2 = (a_1(R_1)(a_4(R_2)a_7)(R_3)a_{10}).$$

Now for every position containing a push symbol in string $x2$ we look for its corresponding pop symbol. For example, the pop symbol a_7 corresponds to push symbol a_4. We replace every push symbol a_i by a pair $[a_i, a_j]$; where a_j is the pop symbol corresponding to a_i, then we erase all pop symbols. The following string is obtained:

$$x3 = ([a_1, a_{10}](R_1)([a_4, a_7](R_2))(R_3)).$$

Finally the operation & is inserted after each pair $[a_i, a_j]$ and the operation $*$ is inserted between each pair of brackets ") (." The operation $*$ has higher priority than &. The resulting expression string is $\mathrm{ex}(x)$. In our case we obtain:

$$\mathrm{ex}(x) = ([a_1, a_{10}] \,\&\, (R_1) * ([a_4, a_7] \,\&\, (R_2)) * (R_3)).$$

The value of the expression $\mathrm{ex}(x)$ is the set of all pairs $((s1, q1), (s2, q2))$ corresponding to the whole input word x. In order to check whether A accepts we verify if a pair $((s_0, q_0), (s_2, q_2))$ is in $\mathrm{ex}(x)$ for initial values s_0 and q_0 of state and top symbol and for some accepting state $q2$. It is easy to see that the transformation $x \to \mathrm{ex}(x)$ can be done by an optimal parallel algorithm. This is because pop and push symbols give the bracket structure of $\mathrm{ex}(x)$ and the correspondence between brackets "(" and brackets ")" can be computed by an optimal parallel algorithm, see Bar-On and Vishkin (1985). The corresponding algebra with operations & and $*$ has finite carrier and our algorithm for the dynamic expression evaluation can be applied. This completes the proof. ∎

Remark. If one wants to remove read conflicts from our algorithm for expression evaluation then it is enough to remove them from Step 1; however, this would imply a substantial improvement of the result of Bar-On and Vishkin. It seems that Step 1 (or its equivalent) is unavoidable.

RECEIVED May 21, 1986; ACCEPTED December 17, 1987

REFERENCES

BAR-ON, I., AND VISHKIN, U. (1985), Optimal parallel generation of a computation tree form, *ACM Trans. Programm. Lang. System* **7**, No. 2, 348–357.

GIANCARLO, R., AND RYTTER, W. (1985), Parallel recognition of input driven and parsing of bracket languages, manuscript, Departement of Informatics, University of Salerno.

KINDERVATER, G., AND LENSTRA, J. (1985), "An Introduction to Parallelism in Combinatorial Optimisation," Report OS-R8501, Centre for Mathematics and Computer Science, Amsterdam.

MATTHEYSES, R., AND FIDUCCIA, C. M. (1982), Parsing Dyck languages on parallel machines, *in* "20th Allerton Conference on Communication Control and Computing."

MILLER, G. L. AND REIF, J. H. (1985), Parallel tree contraction and its applications, *in* "26th IEEE Symposium on Foundations of Computer Science, pp. 478–489."

RYTTER, W. (1985a). The complexity of two way pushdown automata and recursive programs, in "Combinatorial Algorithms on Words" (A. Apostolico and Z. Galil, Eds.), NATO ASI Series F: 12, Springer-Verlag, New York.

RYTTER, W. (1985b), Remarks on pebble games on graphs, presented at "Combinatorial Analysis and Its Applications, Pokrzywna, September 1985; *Zastos. Mat.* in press.

RYTTER, W. (1985c), On the recognition of context free languages, Lecture Notes in Computer Science Vol. 208, pp. 318–325, Springer-Verlag, New York.

RYTTER, W. (1986), An application of Melhorn's algorithm for bracket languages to $O(\log(n))$ space recognition of input driven languages, *Inform. Process. Lett.* **23** (1986), 81–84.

RYTTER, W. (1987), Parallel time $O(\log(n))$ recognition of unambiguous cfls, *Inform. and Comput.* **73**, No. 1, 75–86.