THE UNIVERSITY OF
WARWICK

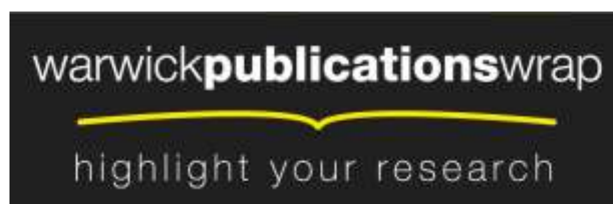
Original citation:
Gibbons, A. M. and Rytter, W. (1986) An optimal parallel algorithm for dynamic expression evaluation and its applications. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-077

Permanent WRAP url:
http://wrap.warwick.ac.uk/60776

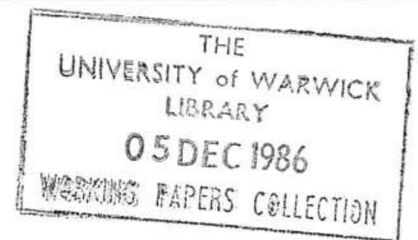
Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at: publications@warwick.ac.uk



http://wrap.warwick.ac.uk/



THE UNIVERSITY of WARWICK LIBRARY
05 DEC 1986
WORKING PAPERS COLLECTION

Research report 77

AN OPTIMAL PARALLEL ALGORITHM FOR DYNAMIC EXPRESSION EVALUATION AND ITS APPLICATIONS

by

Alan Gibbons & Wojciech Rytter*

RR77

(RR77)

Abstract

We describe a deterministic parallel algorithm to compute algebraic expressions in log n time using n/log(n) processors on a parallel random access machine without write confilects (P-RAM) with no free preprocessing. This improves the result of Miller and Reif (1985), who described an optimal parallel randomized algorithm. Our algorithm can be used to construct an optimal parallel algorithm for the recognition of two nontrivial 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), who constructed an almost optimal parallel algorithm for Dyck languages, since Dyck languages are a proper subclass of input-driven languages.

Using our algorithm we show also that preorder and postorder numberings of the nodes of a tree can be computed by optimal parallel algorithms.

Department of Computer Science
University of Warwick
Coventry, CV4 7AL, England

April 1986

* on leave from Institute of Informatics, Warsaw University

AN OPTIMAL PARALLEL ALGORITHM FOR DYNAMIC EXPRESSION EVALUATION AND ITS APPLICATIONS

Alan.Gibbons and Wojciech Rytter*
Dept.of Computer Sc.,University of Warwick
Coventry CV4 7AL, United Kingdom

Abstract:
We describe a deterministic parallel algorithm to compute algebraic expressions in log n time using n/log(n) processsors on a parallel random access machine without write conflicts (P-RAM) with no free preprocessing. This improves the result of Miller and Reif(1985), who described an optimal parallel randomized algorithm. Our algorithm can be used to construct optimal parallel algorithms for the recognition of two nontrivial 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) who constructed an almost optimal parallel algorithm for Dyck languages, since Dyck languages are a proper subclass of input-driven languages.
Using our algorithm we show also that preorder and postorder numberings of the nodes of a tree can be computed by optimal parallel algorithms.

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 RAM's that can access a common memory. Some 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 (i.e., n, n/log(n), n/log^{2}n).Optimal parallel algorithms are known for several simple computational problems. For example, computing associative function of n variables, computing maximum, selection, string matching and converting an expression to its parse tree.

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 given by Rytter(1985). Bar-On and Vishkin(1985) constructed an optimal parallel algorithm (with t=log n, p=n/log n) to convert an expression to its parse tree. Using their result and algorithms of Miller,Reif and Rytter the expression can be computed in log n time with n/log n processors. The key idea is rather simple and has already been used for the computation of associative functions of n variables. In this case use was made of a binary tree of height log n with leaves corresponding to the variables. This tree was evaluated

* On leave from Institute of Informatics,Warsaw University

bottom-up and in order to reduce the number of processors from n to n/log n the leaves were grouped in sections of length log n, and these sections were preprocessed by n/log n processors in log n time. The result of such a preprocessing was a binary tree with n/log n leaves. We follow the same strategy, however now our tree need not to be of logarithmic height and the preprocessing is more involved. The main function of the algorithm is to preprocess the tree during which the number of leaves is reduced. An invariant of the preprocessing is that the trees are binary. Hence the reduction in the number of leaves implies a reduction in the number of all nodes. For the reduced tree we can use almost optimal parallel algorithms of Miller,Reif and Rytter. The whole algorithm is optimal since the size of reduced tree is n/log n.

Bracket languages are context-free languages whose elements are strings with an explicitly encoded structure of the parsing tree. These languages are generated by grammars whose productions are of the form A->(u), where u does not contain 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 (more than this, it also implies an optimal parallel parsing algorithm). Matheyses and Fiduccia(1982) gave an optimal parallel algorithm for the recognition of Dyck languages, which are a proper subclass of input driven languages (id.languages). An almost optimal parallel algorithm for the recognition of these languages was given by Giancarlo and Rytter(1985). Id. languages are generated by very restricted one-way deterministic pushdown automata (dpda's). Each element of the alphabet provides constant changes in the height of the pushdown store. In the case of Dyck languages opening brackets increase the height of the stack by one, and closing brackets cause a decrease. A corresponding dpda can be easily constructed to accept well-composed strings of brackets (with many types of brackets e.g. square and open).

We assume (for ease of exposition) that the height of the stack changes by one. For any id.language accepted by such dpda the elements of the alphabet can be divided into two classes: one corresponding to opening brackets (push) and the other corresponding to closing brackets (pop). The recognition problem 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 are known to be recognizable in log n time is the class of unambiguous cfl's. However in this case we do not anticipate an optimal parallel algorithm (n^7 processors are needed to obtain logarithmic time, see Rytter(1985)). Rytter(1985) 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 id.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.

We can assume that the expression is in binary tree representation since such a representation can be constructed by the optimal parallel algorithm of Bar-On and Vishkin(1985). In this section we allow the arithmetic operations: +,-,*,/. We assume that their cost is O(1). Our construction also works for any algebra with carrier (set of elements) of constant bounded size, which 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 this section is the following:
Theorem 1
(a) Arithmetic expressions can be computed on a P-RAM in log(n) time using n/log(n) processors
(b) algebraic expressions with operations from an algebra with carrier of O(1) size can be computed on a P-RAM in log(n) time using n/log(n) processors.
Proof.

Assume that T is a tree of an arithmetic expression. T has n leaves with associated integer values and each internal node of T has an associated arithmetical operation. It is enough to show how to reduce this tree to an equivalent (corresponding to an expression with the same value) reduced tree RT which has n/log n leaves with integer values and whose internal nodes correspond to certain operations computable in O(1) time. Then RT can be subjected to the algorithms of Miller,Reif and Rytter .
Assume (for ease of exposition) that log n is an integer and n is divisible by log n. This is true for the example of Fig.1, where n=16. This example is used to illustrate an application of the algorithm in the course of which we indicate how the algorithm is applied for the general case.
ALGORITHM

Step 0.
We use the optimal parallel algorithm of Bar-On and Vishkin to convert the expression into its tree representation

Step 1.
Generally, given a tree of an expression, we number the leaves 1..n from left to right. The leaves are then grouped into n/log(n) consecutive segments, each of length log n. For our example this implies 4 segments , each of length 4 as indicated in the caption of Fig.1.
A processor is now assigned to each segment of leaves and conducts a partial evaluation of the expression as follows. Every subexpression which corresponds to a maximal subtree for which every leaf is contained within the same segment is evaluated, and the tree is modified by reducing this subtree to a single leaf vertex to which the value of the subexpression is assigned. In our

example each segment contains just one subtree (generally of course it can be any number) and these are indicated in Fig.1 whilst the modified tree is shown in Fig.2. It is easy to see that this step can be achieved in O(log n) parallel time as follows. A vertex is called a vertex of type "/" if it is a left son of some vertex, similarly a right son is called a vertex of type "\". If each vertex in the representation of the tree has a record of its type, then we can use a simple stack and a single pass of the log n leaves in a segment to achieve our aim. The type of each leaf simply plays the part of the bracket, so that the subtrees can be reduced in an obvious way.

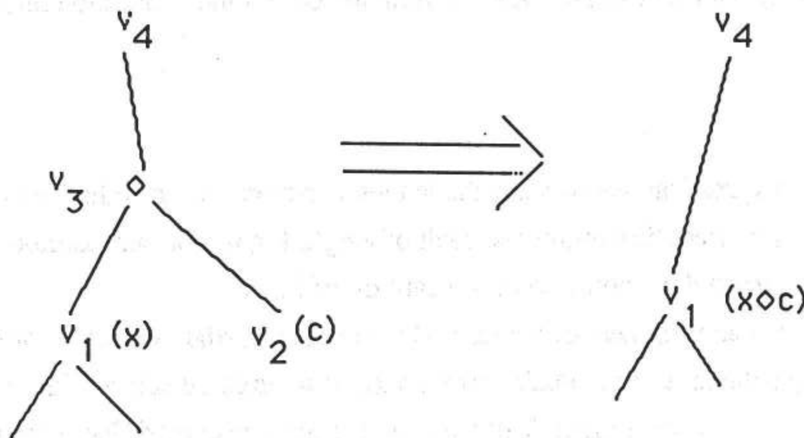
It is easy to see that after reduction of these subtrees, the resulting leaves in any segment (if we represent the leaves by their types) can form strings only of the kind \\.....\//../ $=\backslash^i/^j$, where it is possible that one of i and j is zero. For step 2 of the algorithm we call the first leaf of type "\" and the last leaf of the type "/" in this sequence *nonreducible*. All other leaves are called *reducible*. Observe that generally there are at most O(n/log(n))) nonreducible leaves.

Step 2

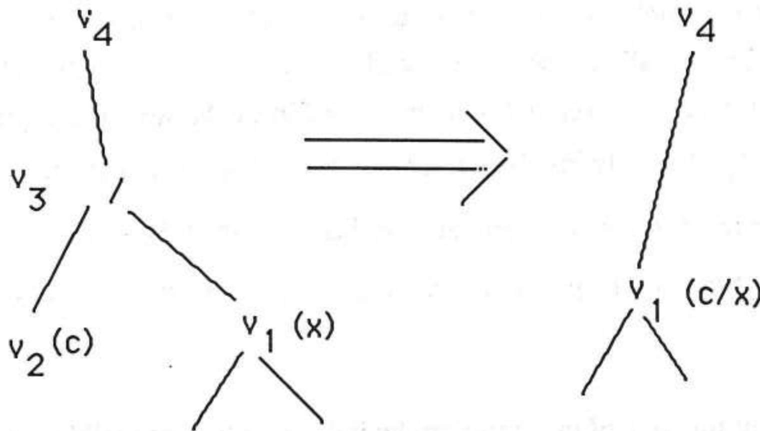
This step of the algorithm reduces the tree further by removing all reducible leaves. We describe a parallel means of doing this, but essentially such a leaf is removed by local reconstruction of the tree.

A functional form (an algebraic expression) $f_i(x)$ is associated with each 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 associated values c1 and c2, then the value of v_i is given by substituting $(c1 \Diamond c2)$ for x in $f_i(x)$.

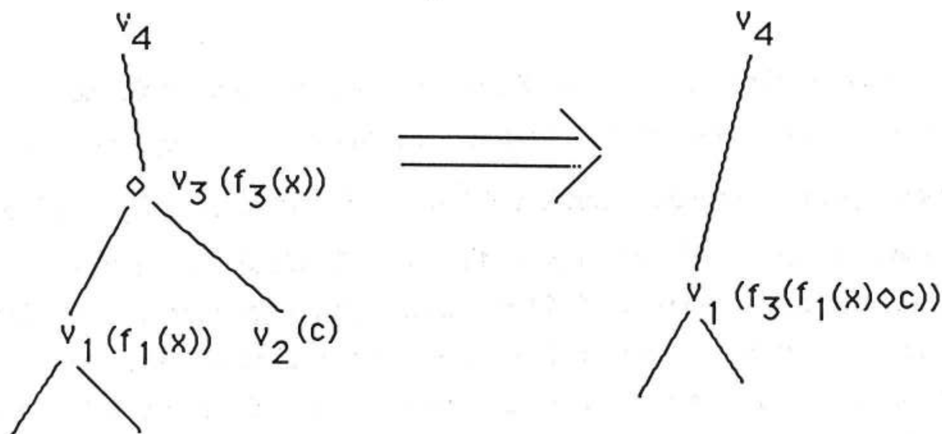
Consider removing a single reducible leaf and imagine for the moment that no other such leaves have been removed. If v_2 is the leaf (with an associated constant value c) to be removed and if father(v_2) is not the root of the tree, then the tree is reconstructed locally as follows:



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 stored at v_1 is of course the value of the subtree originally rooted at v_3 and so the new tree computes the same expression as the original tree. If the operator at v_3 is noncommutative, the functional form stored at v_1, after reconstruction, properly reflects this. For example:



If father(v_2) is the root of the tree then in these examples v_4 is removed from the illustrations. Thus we have described the removal of a first and just one reducible 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 independent reductions. 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 (independent) reductions. We do this schematically as follows:



Again, without loss of generality, father(v_2) is presumed not to be the root of the tree .
A crucial observation concerning the removal of a reducible leaf is that it can be achieved in constant bounded time. Consider first 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 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 implied in the last figure) 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 constant time (by movement of father, son pointers).

We complete the description of this step of the algorithm by indicating how reducible leaves are removed in parallel. In the previous step of the algorithm a processor was assigned to each of n/log(n) segments of leaves, reducing each to a string of the form $\backslash^i/^j$. Each processor now (and in parallel with other processors working on their respective segment) first removes reducible leaves of the type "/". It does this in a single left-to-right pass of the leaves in this segment. On completion of that parallel operation the processors now in parallel remove reducible leaves of type "\" in a single right-to-left pass of their segments. Since individual reductions can be achieved in constant time and segments are of length at most log(n), this step of the algorithm requires O(log n) time. Notice that the order of leaf removal described ensures that removals are carried out independently. Figs.2,3 and 4 illustrate this step of the algorithm for our example.

<u>Step</u> 3
The output from step 2 of the algorithm is a binary tree RT with O(n/log(n)) leaves and hence the number of vertices is also of the same order. Each nonleaf vertex v_i has an associated functional form $f_i(x)$ and an arithmetic operator. Now the algorithm of Miller and Reif(1985) or Rytter(1985) can be used to compute the value associated with the root of the tree RT. We shortly describe Rytter's construction (in fact this one and the one of Miller and Reif are very similar, though using quite different terminologies). The computation is "pebble-driven", a special parallel pebble game is defined. Associate with each node v a node cond(v), initially cond is the identity function.The pairs (v,cond(v)) can be treated as additional edges (Miller and Reif use the function P for similar purposes).

Computing the complete information associated with a node corresponds to pebbling the node. The nonleaf node v can be pebbled iff we know some information corresponding to the edge (v,cond(v)) (a functional form expressing the value of v in terms of the value corresponding to cond(v)) and the node cond(v) is pebbled (has completely computed information).
The parallel pebble game is defined for binary trees (it can be defined for any tree with constant bounded degree but then it needs slight modification when defining the operation activate).

We say that a node v is active iff cond(v)≠v.
We define three operations activate,square and pebble as follows:

activate:
<u>for each</u> nonleaf node v <u>do in parallel</u> if v in not active and one of its sons is pebbled then set cond(v) to the other son , if both sons are pebbled then choose one of them {v becomes active};

square:
<u>for each</u> node v <u>do in parallel</u> cond(v):=cond(cond(v));

pebble:
<u>for each</u> node v <u>do in parallel</u> if cond(v) is pebbled then pebble v;

Let one pebbling move consist of executing the sequence of operations (activate;square;square;pebble), in that order. It was proved by Rytter (1985) that if T is a binary tree with m leaves, all initially pebbled, then after applying $\lceil \log m \rceil$ pebbling moves the root is pebbled.
We can associate a functional form F_v with each node v, which reflects the dependence of the value associated with v on the value associated with cond(v). Initially F_v=x (identity). However the functional forms associated with nodes in step 2 had a different use and meaning than the functional forms F , which are in fact associated with edges (v,cond(v)). All the forms which can appear can be written in the form ((ax+b)/(ex+d)).
Whenever we execute the operation square we set F_v to $F_v(F_{cond(v)})$ just before executing cond(v):=cond(cond(v)), and whenever we execute the operation activate we compute F_v using the value of one of sons of v (already known) and the operator and functional form associated with v obtained in step 2. For example if the left son is not pebbled, the right son of v is pebbled and its corresponding value is 7 ,suppose the operator is / and the functional form (associated with v in step 2) is 3*x+5 ,then the functional form obtained (and corresponding to (v,leftson(v)) is :

$F_v(x) = 3*(x/7)+5$.

This form means that if the value associated with cond(v) is x then the value associated with v is 3*(x/7)+5. Now we can play our parallel pebble game and simultaneously compute values associated with nodes and functional forms associated with edges (v,cond(v)). This additioanl computing is done in the same manner as by Miller and Reif(1985), we refer the reader there for details. The number of processors used is proportional to the number of the nodes of the tree. Since our tree RT has O(n/log(n)) nodes the required number of processors is n/log(n). After log n pebbling moves (together with accompanying computations of values and functional forms) the value corresponding to the root (and the whole expression) is evaluated.
end of algorithm.

When the input expression is an algebraic expression with operations from an algebra with a constant number of elements, a slight modification is required. Instead of functional forms we use functions, whose values and arguments are elements of the algebra. Such functions will reflect the dependance of the value associated with one node on the value associated with another node (in step 3), or an additional action to be made when computing a value associated with the node (in step 2). The description of such 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. □
Remark. It is possible to merge steps 0 and 1 of the algorithm.

3. Applications

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 iff each production is of the form A->(u), where u does not contain brackets "(", ")".
For ease of exposition assume that G is in a Chomsky-like Normal Form, each production being of the type A->(BC), or A->(a), where B,C are nonterminals and a is a terminal.
By a recognition problem we mean the following problem: given an input string w and a grammar, decide whether w is generated by the grammar.
The size of the problem is the length n of the input string w.
Any specific recognition problem can be reduced to the evaluation of a certain algebraic expression. Define the operation * on sets of nonterminals as follows:
S1*S2= {A : A->BC is a production and B∈S1, C∈S2 }.
Consider the following example:
S->(BC) S->(SA) A->(BB) A->(a) B->(AC) B->(b) B->(a) C->(BB).
Let the input string be
w = ((a)(((b)(b))(a))).

Replace each substring (x) in w by a set {A: A->(x)}, where x is terminal symbol, next replace each substring (S1 S2) ,) S1), (S1 (, by (S1*S2),)*S1), (S1*(, respectively, where S1, S2 are sets of nonterminals. For the example string we obtain :
({A,B}*(({B}*{B})*{A,B})).
The obtained string is an algebraic expression. In the example the value of the expression is {A,S}. It can be easily seen that the input string w is generated by a grammar iff 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 the carrier of O(1) size. Hence Theorem 1 implies:
<u>Theorem</u> 2
The recognition problem for bracket languages can be solved on a P-RAM in log(n) time using n/log(n) processors.

Let L be an id.language accepted by a deterministic pushdown automaton A.
Each symbol of the alphabet corresponds to a push or a pop move. We can assume (for ease of presentation) that one symbol is pushed. Now push symbols correspond to opening brackets and pop symbols correspond to closing brackets. Braunmuhl,Verbeek(1983) provide more details about dpda's accepting id.languages.
The recognition problem can be again reduced to the problem of computing some algebraic expression. The carrier of the corresponding algebra is the set of all subsets of (S©W)©(S©W), where © denotes Cartesian product of sets, S is the set of states and W is a pushdown alphabet of the automaton A. The value corresponding to a substring v of the input string is the set
{ ((s1,q1),(s2,q2)) : automaton A starting in the state s1 with q1 as the only element of its stack after reading v can be 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 by Rytter(1984) to design a space efficient algorithm. We refer the reader there for details.
The automaton A accepts the input string iff the value corresponding to the whole string contains an element of the form ((s0,q0),(s',q')) , where s0 is an initial state, q0 is an initial element of the stack and s' is an accepting state. The brackets corresponding to pop and push symbols describe the structure of the corresponding algebraic expression. There are two operations: composition of relations and a very technical unary operation corresponding to stack operations, see Rytter(1984). The nodes associated with the operation of composition can in fact have an unbounded number of sons . However this difficulty can be easily avoided in this special case by restructuring the expression, the details are omitted. Hence again the recognition problem for L can be reduced to a computation of an algebraic expression and the carrier of the algebra used here has O(1) size. As an application of Theorem 1 we obtain:

Theorem 3

The recognition problem for id.languages can be solved on a P-RAM using an optimal parallel algorithm.

Let • be any associative opertation computable on a RAM in O(1) time. Assume that each edge x of the binary tree T is associated with a value val(x). Define the path problem to be the problem of computing for each node v the value $f(v)=val(x_1)\bullet val(x_2)\bullet ... val(x_k)$, where $x_1,...,x_k$ are all the edges on the path from v to the root. The path problem is similar to the type-1 tree function defined by Huang(1985), the only difference is that we associate values with edges rather than with nodes.

Theorem 4

The path problem can be computed on a P-RAM in log(n) time using n/log(n) processors.

Proof.

We can use essentially the same preprocessing as in the proof of Theorem1. After preprocessing we obtain a reduced version RT of T .Some nodes of this tree can have additional values resulting from a compression of some path. We can compute the required result f(x) for all nodes of the reduced tree RT using a doubling technique similar to that used by Huang(1985) to compute type-1 tree functions. Now we can compute f(x) for all nodes eliminated during the preprocessing by applying a process which is the reverse of the preprocessing. This requires some additional bookkeeping during the preprocessing. The details are very technical and we omit them. □

Theorem 5

The preorder and postorder numberings of a binary tree can be computed on a P-RAM in log(n) time using n/log(n) processors.

Proof.

Let nd(x) denote the number of descendants of the node x. We can compute the function nd(x) for each node using the algorithm of theorem 1. The tree can be treated as a tree of an expression. Each leave has the value 1, and each operation in the expression is the operation x+y+1 (summing numbers of nodes in the left and the right subtrees plus one,which includes the node itself).

For each edge (x,y) where x is the father of y, we set

val(x,y)= if y is the right son then nd(left son of x) else 0.

Let f(x) be the sum of values of edges on the path from x to the root. We can compute f by an algorithm from theorem 4. The postorder number of the node x is now nd(x)+f(x).

The preorder number can be computed similarly summing f(x) and the number of nodes on the path from x to the root (the last number can be computed using essentially the same algorithm as in the proof of theorem4, but now we sum values of nodes on the path to the root). For all nodes we have to sum two values in O(1) time using n processors or in log(n) time using n/log(n) processors (grouping the nodes).

The whole algorithm has the required complexity. This completes the proof. □

References.

I.Bar-On,and U.Vishkin.(1985), Optimal parallel generation of a computation tree form. ACM trans.on Progr.Lang.and Systems 7,2,348-357

B.Braunmuhl,and R.Verbeek(1983),Recognizing input-driven languages in log n space, Fund.of Comp.Theory,Lect.Notes in Computer Science

R.P.Brent.(1974), The parallel evaluation of general arithmetic expressions. JACM 21,2, 201-208

R.Giancarlo,and W.Rytter(1985), Parallel recognition of input driven and parsing of bracket languages. manuscript, University of Salerno,Dep. of Informatics

Ming-Deh A.Huang (1985), Solving some graph problems with optimal or near optimal speed up on mesh-of-trees networks. 26th IEEE Symp.on Found.of Comp.Science, 232-240

R.Mattheyses,and C.M.Fiduccia (1982), Parsing Dyck languages on parallel machines. 20th Allerton Conference on Comm.Control and Computing

G.L.Miller,and J.Reif (1985), Parallel tree contraction and its application. 26th IEEE Symp. on Found.of Comp.Science,478-489

W.Rytter(1984),An application of Mehlhorn's algorithm for bracket languages to log n space recognition of input driven languages, Berichte Theoretische Informatik, Fachb.Math.Informatik,Universitat Paderborn (to appear in Inf.Process.Letters)

W.Rytter (1985),Parallel time O(log n) recognition of unambiguous cfl's. Fund.of Computation Theory,Lect.Notes in Comp.Science

W.Rytter (1985), The complexity of two way pushdown automata and recursive programs. in Combinatorial algorithms on words (ed.A.Apostolica,Z.Galil), NATO ASI Series F:12, Springer Verlag

W.Rytter (1985), Remarks on pebble games on graphs. in Combinatorial analysis and its applications (ed.M.Syslo), to appear in Lect.Notes in Math.

W.Rytter(1985),On the recognition of context free languages, Lect.Notes in Comp.Science 208

R.E.Tarjan,and U.Vishkin (1984),Finding biconnected components and computing tree functions in logarithmic parallel time.25th Symp.on Found. of Computer Science,IEEE,12-22

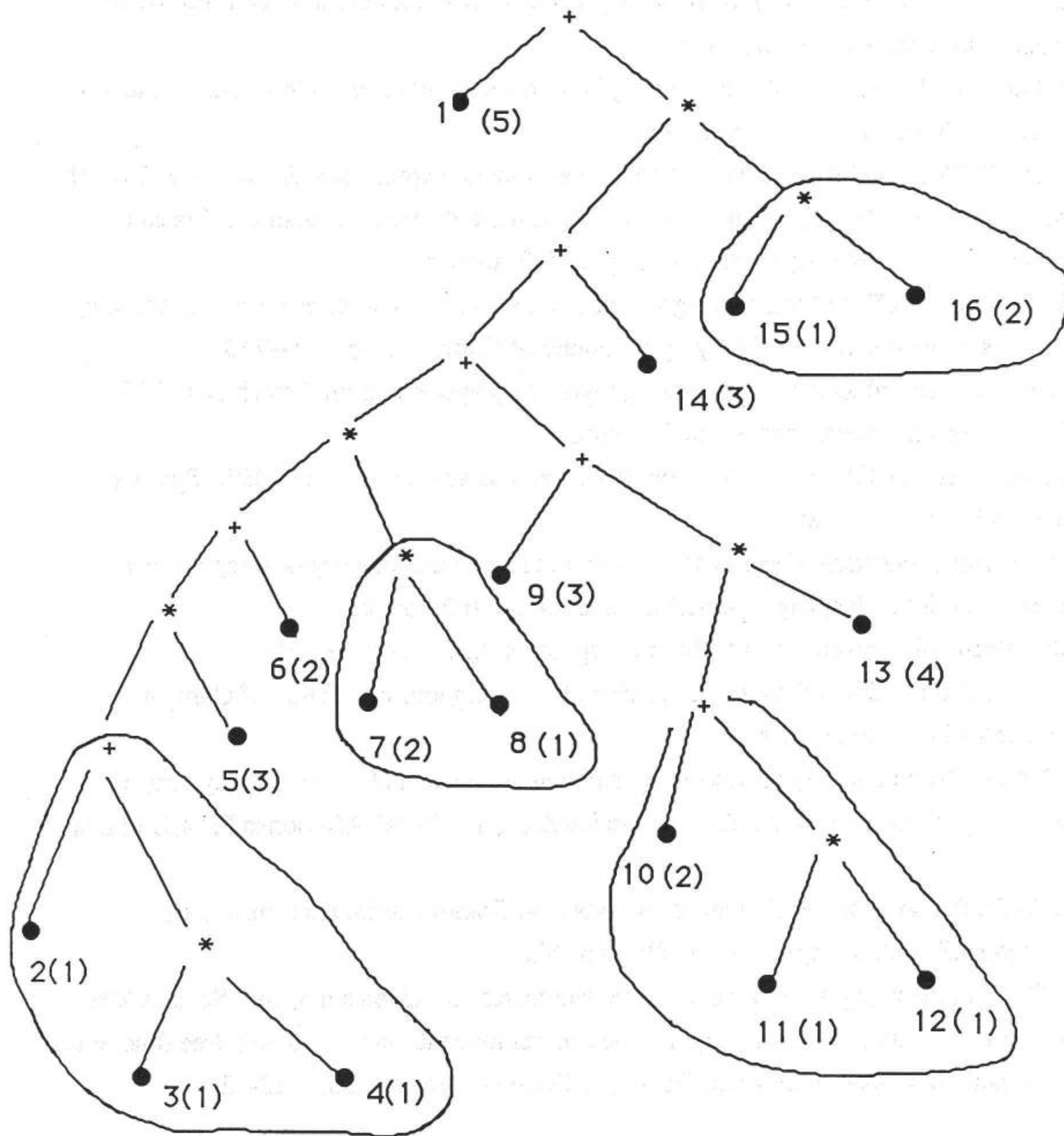


Fig. 1. The tree of an expression. The values of leaves are placed in brackets. There are 4 segments of leaves. [1,2,3,4][5,6,7,8][9,10,11,12][13,14,15,16]. The indicated subtrees are reduced in step 1.

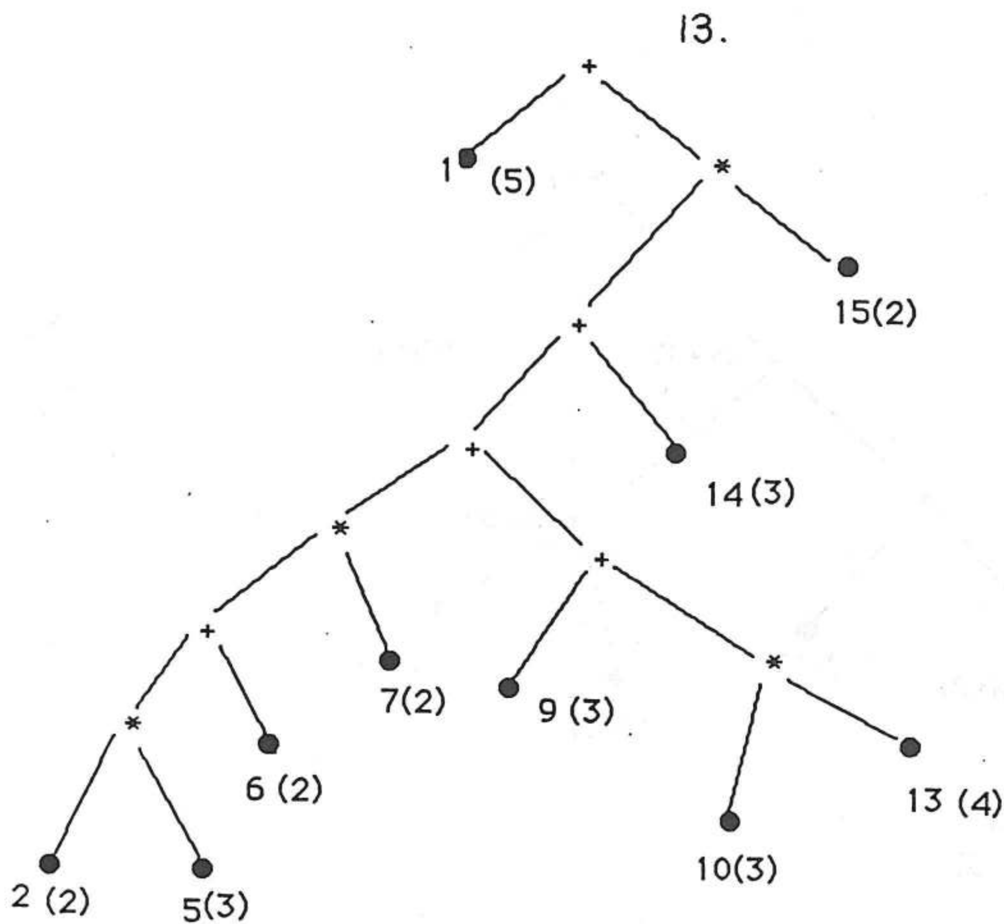


Fig.2. The tree after step 1. We have three groups of leaves [1,2] [5,6,7] [9,10] [13,14,15]. The leaves are of the types [/,/] [\,\,\] [/,/] [\,\,\]. We shall reduce leaves 1,9, next leaves 6,14, and next 7,15.

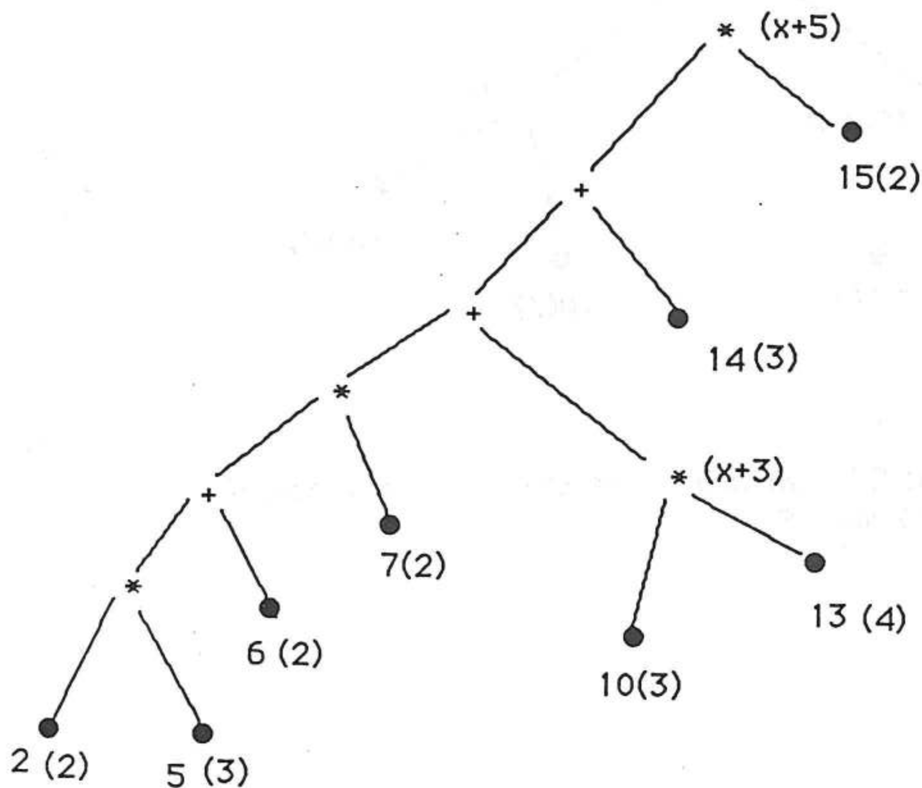


Fig.3. The tree after removing (simultaneously) leaves 1 and 9.

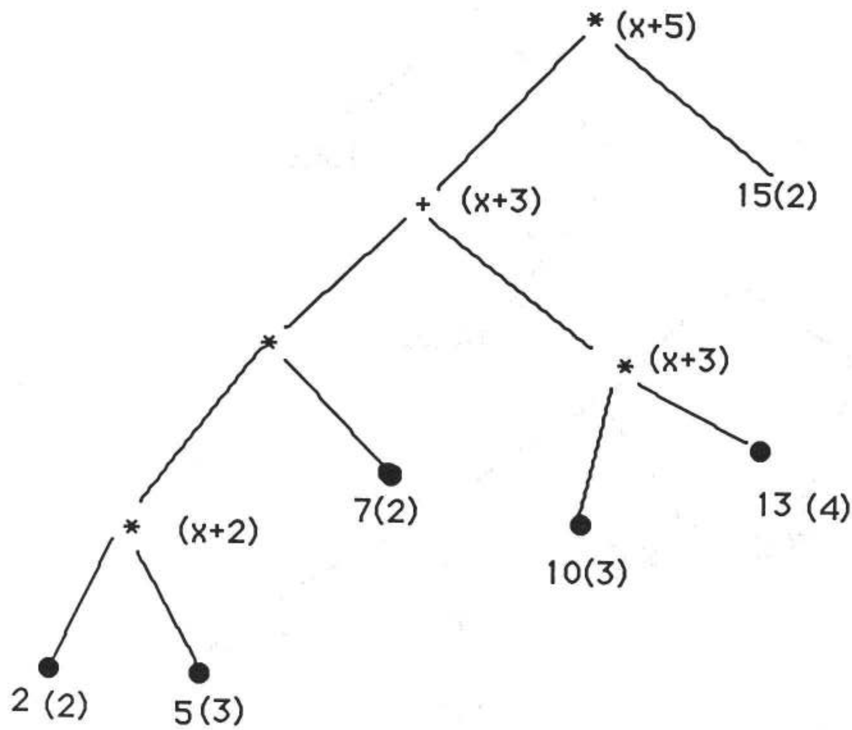


Fig.4. The tree after removing (simultaneously) leaves 3 and 14

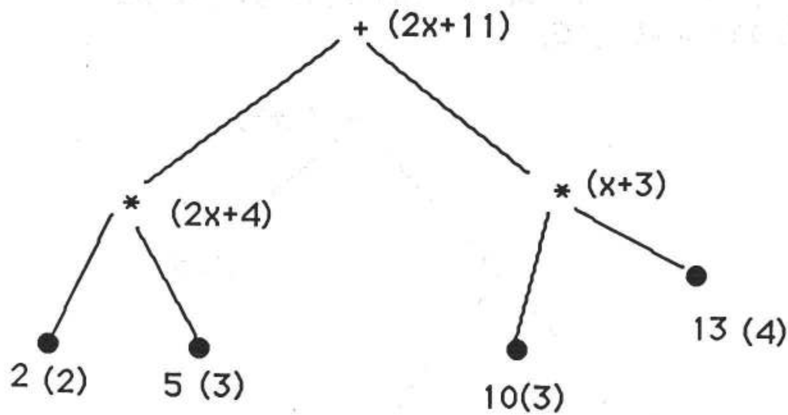


Fig.5. The tree RT results after removing (simultaneously) leaves 7 and 15.