

**Original citation:**

Rytter, W. and Giancarlo, R. (1985) Optimal parallel parsing of bracket languages. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-085

**Permanent WRAP url:**

<http://wrap.warwick.ac.uk/60781>

**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.

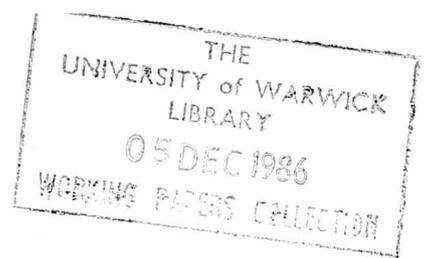
Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is 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](mailto:publications@warwick.ac.uk)



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



# Research report 85

## OPTIMAL PARALLEL PARSING OF BRACKET LANGUAGES

Wojciech Rytter\*

&

Raffaele Giancarlo†

(RR85)

### Abstract

We prove that the parsing problem for bracket context-free languages can be solved in  $\log(n)$  time using  $n/\log(n)$  processors on a parallel random access machine without write conflicts (P-RAM). On the way we develop a new general technique for tree compression based on the bracket structure of the tree.

Department of Computer Science  
University of Warwick

\* and Institute of Informatics, Warsaw University, Poland

† Dept of Informatics, University of Salerno, Italy and  
Dept of Computer Science, Columbia University, New  
York, USA

Oct 1985



## OPTIMAL PARALLEL PARSING OF BRACKET LANGUAGES

Wojciech Rytter

Dept. of Computer Science, University of Warwick, Coventry CV4 7AL, U.K.  
and Institute of Informatics, Warsaw University, Poland

Raffaele Giancarlo

Dept. of Informatics, University of Salerno, Italy  
and Dept. of Computer Science, Columbia University, New York, USA.

**Abstract:** We prove that the parsing problem for bracket context-free languages can be solved in  $\log(n)$  time using  $n/\log(n)$  processors on a parallel random access machine without write conflicts (P-RAM). On the way we develop a new general technique for tree compression based on the bracket structure of the tree.

An optimal parallel algorithm (for a given problem computable sequentially in linear time) is one that satisfies  $p \cdot t = O(n)$ , where  $p$  is the number of processors used,  $t$  is the parallel time and  $t$  is very small (i.e.,  $\log(n)$ ,  $\log^2 n$ ). Optimal parallel algorithms are known for few nontrivial computational problems: computing associative function of  $n$  variables, selection, string matching, converting an expression to its parse tree, dynamic evaluation of expressions. We add to this list another problem: parsing bracket languages. In some sense it can be treated as a generalization of optimal conversion of expressions into parse trees (if we assume that expressions are fully parenthesized).

Bracket languages are one of the few interesting subclasses of context-free languages known to be recognizable sequentially in logarithmic space, it is natural to expect that a problem easy with respect to the space complexity could be also solvable by an efficient parallel algorithm. Another subclass of context-free languages recognizable in logarithmic space is the class of input driven languages [10]. For such languages it was also shown that an optimal parallel algorithm for the recognition problem is possible, see [2].

For bracket languages we give a much stronger result, since we deal with the parsing problem which seems to be more difficult than the recognition problem. As far as we know the only nontrivial context-free languages for which an optimal parallel parsing algorithm is (implicitly) known are Dyck languages (well formed sequences of constant types of brackets). In fact the algorithm given in [5] is not an optimal parallel algorithm, since  $p = n$  and  $t = \log(n)$  in [5]. However the number of processors can be reduced to  $n/\log(n)$  using the result of [1] in a straightforward way. One can disregard types of brackets and compute for each bracket its corresponding matching bracket using the algorithm from [1], then the types of matching pairs can be checked. The same trick is used in  $\log n$  space recognition of Dyck languages.

Our model of parallel computation is a parallel random access machine without write conflicts (P-RAM). Such a model is known also as a CREW P-RAM. It consists of a number of synchronously working processors (RAM's) which are using a common memory. No two processors can attempt to write in the same step into the same location, however many processors can read from the same location. Such a model corresponds to bounded fan-in circuits.

The best algorithms for parallel general context-free recognition on a P-RAM work in  $\log^2(n)$  time using  $O(n^6)$  processors, see [11] (such complexity can even be achieved on much weaker models of parallel computations, cube connected computers and perfect shuffle computers, see [9]). For unambiguous languages  $\log n$  time is enough, however the number of processors is bounded by a polynomial with a high degree [12].

Recently it was proved in [2] that bracket languages can be recognized using an optimal parallel algorithm ( $\log n$  time and  $n/\log(n)$  processors).

We show that the parsing problem for these languages can be also solved using an optimal parallel algorithm.

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  $A \rightarrow (u)$ , where  $u$  does not contain brackets "(", ")".

A language is a bracket language iff it is generated by a bracket grammar. A typical example of a bracket language is the set of parathesized arithmetic expressions with constants  $a, b$ . It is generated by the grammar:

$$E \rightarrow (E) \mid (E * E) \mid (E + E) \mid (E - E) \mid (E / E) \mid (a) \mid (b)$$

The text generated by a bracket grammar contains explicit information about the "shape" of the parsing tree (given by the bracket structure of the text). However such information does not give directly the full information about the parsing tree, the labels (nonterminals) associated with the nodes are missing. The aim of this paper is to prove that the computation of these labels can be done by an optimal parallel algorithm. The number of possible correct labellings can be exponential and, though for each text the shape of its parse tree is uniquely determined, the grammar can have a very large degree of unambiguity. For example consider the grammar

$$S \rightarrow (SA) \mid (AS) \mid (SS) \mid (AA) \quad A \rightarrow (a).$$

We are concerned with finding any correct labelling (a parse tree if there is any).

For ease of exposition assume that  $G$  is in a Chomsky-like Normal Form, each production being of the type  $A \rightarrow (BC)$ , or  $A \rightarrow (a)$ , where  $B, C$  are nonterminals and  $a$  is a terminal. Our method can be extended to arbitrary bracket grammar. Instead of binary trees one has to consider trees with degrees bounded by a constant.

We write  $A \rightarrow^* w$  iff the string  $w$  can be derived from  $A$ . Let  $S$  be the starting symbol of the grammar. Let  $w$  be a given input string of length  $n$ . The parsing problem is: construct a parsing tree  $PT$  (if  $S \rightarrow^* w$ ). The size of the problem is  $n$ .

(We assume that the grammar is given and has a constant size independent of  $n$ .)

The tree  $PT$  is represented by arrays whose entries correspond to the nodes of the tree. With each

node  $x$  there is associated information about its sons, father and label. The label of the root is  $S$ .

### Example

Consider the grammar:

$S \rightarrow (BA) \mid (BC) \mid (AS)$

$A \rightarrow (BA) \mid (AA) \mid (BB) \mid (a)$

$B \rightarrow (CC) \mid (CS) \mid (b)$

$C \rightarrow (AB) \mid (CA) \mid (BC) \mid (a)$

and the input text

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

The bracket structure of  $w$  is:

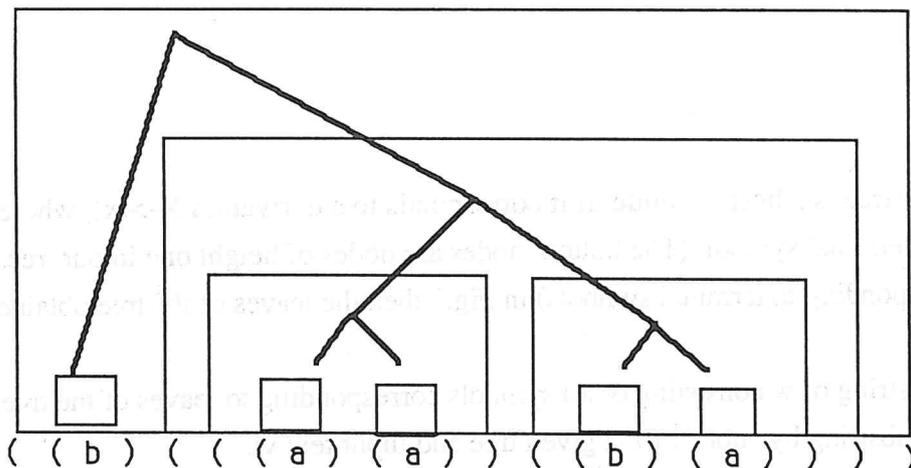


Fig.1

Each node corresponds to a pair of matching brackets. The sons of a given pair are pairs enclosed within it. This determines the shape of the parse tree, which is shown in Fig.1. The nonterminals corresponding to internal nodes of such a tree are at present unknown. Their computation by an optimal parallel algorithm is the aim of our paper.

It was shown in [1] that such a tree can be constructed by an optimal parallel algorithm. At this stage (computing the "shape" of the tree) all symbols except brackets can be ignored.

### Lemma 1 (Bar-on, Vishkin)

The tree corresponding to the sequence of brackets can be constructed in  $\log(n)$  time using  $O(n/\log(n))$  processors on a P-RAM. It can also be checked with the same complexity whether the sequence is a well formed sequence of brackets.

Define the operation  $*$  on sets of nonterminals as follows:

$S_1 * S_2 = \{A : A \rightarrow (BC) \text{ is a production and } B \in S_1, C \in S_2\}$ .

For our example grammar we have  $\{A, B, C\} * \{B, C\} = \{A, S, C\}$ .

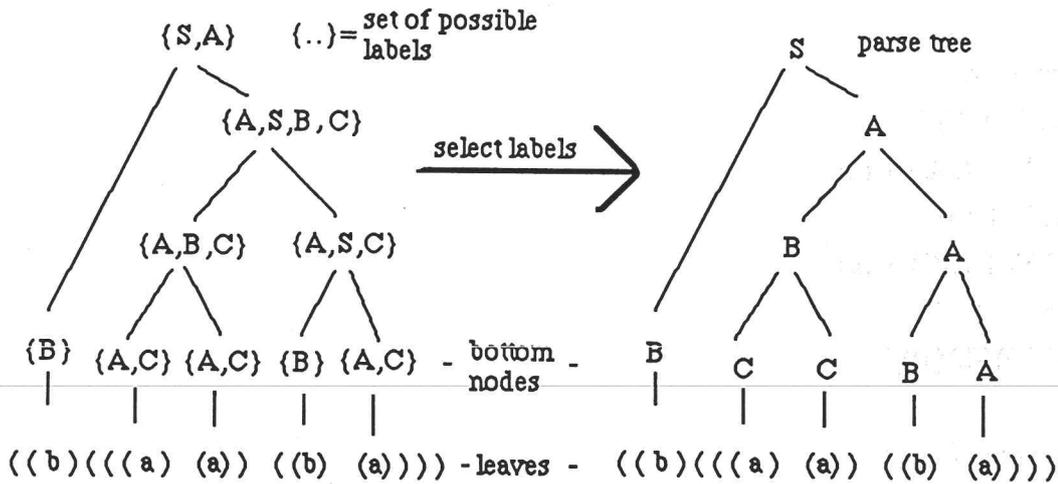


Fig.2

We say that a node in our tree is a **bottom node** if it corresponds to a derivation  $X \rightarrow (x)$ , where  $X$  is a nonterminal and  $x$  is terminal symbol. (The bottom nodes are nodes of height one in our tree, if we remove leaves (corresponding to terminal symbols) in Fig.2 then the leaves of the tree obtained will be bottom nodes.)

Each node  $v$  covers a substring of  $w$  consisting of all symbols corresponding to leaves of the tree rooted at  $v$ . Denote this substring by  $\text{sub}(v)$ , for a given tree and input text  $w$ .

For each internal node  $v$  define  $\text{val}(v) = \{X : X \rightarrow^* \text{sub}(v)\}$ .

It is easy to compute  $\text{val}(v)$  for each bottom node, it takes  $O(1)$  time per one node since the size of the grammar is constant. For a bottom node  $v$   $\text{sub}(v) = "(x)"$  for some terminal symbol  $x$ , and we have to find all nonterminals  $X$  such that  $X \rightarrow (x)$  is a production. Hence  $\text{val}(v)$  can be computed for all bottom nodes in  $O(1)$  time using  $n$  processors or in  $\log(n)$  time using  $n/\log(n)$  processors.

If  $v$  is an internal node which is not a bottom node then it has two sons. Denote them by  $v_1$  and  $v_2$ . It follows from the definition that  $\text{val}(v) = \text{val}(v_1) * \text{val}(v_2)$ . Hence the problem of computing  $\text{val}(v)$  for all  $v$  turns out to be the problem of computing values of all nodes in a tree of an algebraic expression (assuming that the values of bottom nodes are already computed). The underlying algebra is not very regular. For example the operation  $*$  can be nonassociative. However the carrier of the algebra is finite. It was proved in [2] that the computation of  $\text{val}$  can be done by an optimal parallel algorithm.

### Lemma 2 (Gibbons, Rytter)

Let  $T$  be the tree of an algebraic expression. If the leaves of  $T$  are consecutively numbered from left to right and the carrier of the underlying algebra has cardinality bounded by a constant then  $\text{val}(v)$  for each node  $v$  can be computed in  $\log(n)$  time using  $n/\log(n)$  processors on a P-RAM.

In our case the leaves of the expression (bottom nodes) can be consecutively numbered in  $\log(n)$  time using  $n/\log(n)$  processors. Each bottom node  $v$  corresponds to a position  $i$  containing a terminal

symbol  $x$ . Such positions can be easily numbered by assigning 1 to each position with a terminal symbol which is not a bracket and 0 to other positions. Now for a given position  $i$  we can compute the sum of all the assigned integers to the left of  $i$  (including  $i$ ). This gives the correct number for the bottom node  $v$  corresponding to  $i$ . Such a computation is a classical prefix computation and can be performed by an optimal parallel algorithm organizing the processors in the regular binary tree, see [4].

Hence we can assume now that  $\text{val}(v)$  is computed for each node  $v$ .

Now we have to choose one nonterminal from each set  $\text{val}(v)$ . We cannot make this choice locally, since many conflicts (with respect to the grammar) would occur.

We associate with each node  $v$  a partial function  $D_v$ , called the dependancy function. The arguments and values of this function are nonterminals. The interpretation of  $D_v(A)=B$  is: if  $\text{label}(\text{father}(v))=A$  then  $\text{label}(v)=B$ .

Next we execute the following algorithm.

for each internal non-bottom node  $v$  do in parallel

    let  $v_1, v_2$  be the left and right son of  $v$ , respectively ;

    for each  $X \in \text{val}(v)$  choose  $Y \in \text{val}(v_1)$  and  $Z \in \text{val}(v_2)$  such that  $X \rightarrow (YZ)$  is a production;

$D_{v_1}(X) := Y; D_{v_2}(X) := Z; \{ \text{invariant: suitable } Y, Z \text{ can always be found} \}$

### Example

Let  $\text{val}(v) = \{A, S, B, C\}$ ,  $\text{val}(v_1) = \{A, B, C\}$ ,  $\text{val}(v_2) = \{A, S, C\}$ .

Then for  $A$  we can choose  $B \in \text{val}(v_1)$ ,  $A \in \text{val}(v_2)$ , since  $A \rightarrow (BA)$  is a production.

We set  $D_{v_1}(A) = B, D_{v_2}(A) = A$ . For  $S \in \text{val}(v)$  we can choose also  $B, A$  and set  $D_{v_1}(S) = B,$

$D_{v_2}(S) = A$ . Analogously we can set  $D_{v_1}(B) = C, D_{v_2}(B) = C$  and  $D_{v_1}(C) = C, D_{v_2}(C) = A$ .

The function  $D_{v_1}$  can also be written in the form

$A \rightarrow B$

$S \rightarrow B$

$B \rightarrow C$

$C \rightarrow C$ .

The functions associated with each node written in such a form are presented below for our example tree.

The functions  $D_v$  for all nodes  $v$  can be easily computed in  $O(1)$  time with  $n$  processors or in  $O(\log n)$  time with  $n/\log(n)$  processors.

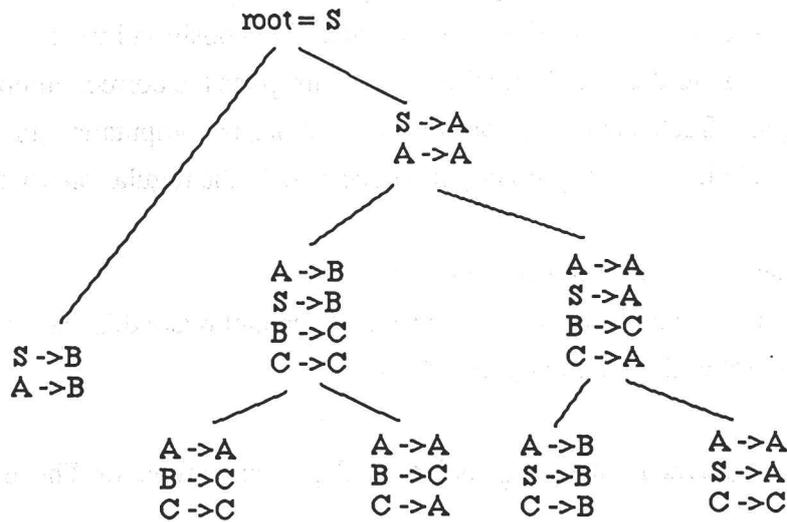


Fig.3.  $A \rightarrow B$  means here: if the father has the label  $A$  then this node should have the label  $B$ .  
We know that the root should have label  $S$ . What are labels implied for other nodes?

#### Remark

Observe that we resolve the (possible) unambiguity of the grammar when computing the functions  $D_v$ . We have many possibilities to choose corresponding nonterminals and we fix one of them. It might seem that there are write conflicts, however this is not so. Each local search is in a set of possibilities (pairs  $Y, Z$ ) whose cardinality is bounded by a constant. The choice can be made sequentially for a given node in  $O(1)$  time, and determinism can be achieved by selecting (for example) the pair corresponding to a production with the smallest number.

Whenever we compute  $D_v$  then  $\text{val}(v) = \text{val}(v_1) * \text{val}(v_2)$ . This guarantees that for each  $X \in \text{val}(v)$  there are suitable  $Y \in \text{val}(v_1), Z \in \text{val}(v_2)$ .

If we require that the root has the label  $S$  then the functions  $D_v$  determine uniquely the label for each node in a top-down way. The value in the root determines the values in the sons of the root through their functions  $D$ , this determines values for their sons etc.

Define the function composition  $f \circ g(x) = g(f(x))$ .

For each internal nonroot node  $v$  let  $F_v = f_1 \circ f_2 \circ \dots \circ f_k$ , where  $f_1, \dots, f_k$  are functions  $D$  associated with the nodes on the path (top-down) from the root to  $v$  (excluding the root and including  $v$ ).

For each node  $v$  in parallel we set  $\text{label}(v) = F_v(S)$ . This gives the full parse tree.

The parse tree determined by the functions in Fig.3 is shown in Fig.2. The selection of labels can be done in  $O(1)$  time with  $n$  processors, or in  $\log(n)$  time with  $n/\log(n)$  processors if the values of  $F_v$  are already computed for each  $v$ .

Hence the parse tree can be constructed by an optimal parallel algorithm if the functions  $F_v$  can be computed by an optimal parallel algorithm

### Theorem

Every bracket language can be parsed in  $\log(n)$  time using  $n/\log(n)$  processors on a P-RAM.

### Proof.

It is enough to show that the functions  $F_v$  for all internal nonroot nodes  $v$  can be computed in  $\log(n)$  time using  $n/\log(n)$  processors. One possibility is to extend the method used in [2] for optimal dynamic evaluation of expressions. However we develop a slightly different method, which is more suitable in this case.

We first show how to compute the functions  $F_v$  in  $\log(n)$  time using  $n$  processors. The method uses the doubling technique. Assume that  $\text{father}(\text{root})=\text{root}$  and  $F_{\text{root}}=\text{identity}$  function. Initially for each nonroot node  $v$   $F_v=D_v$ .

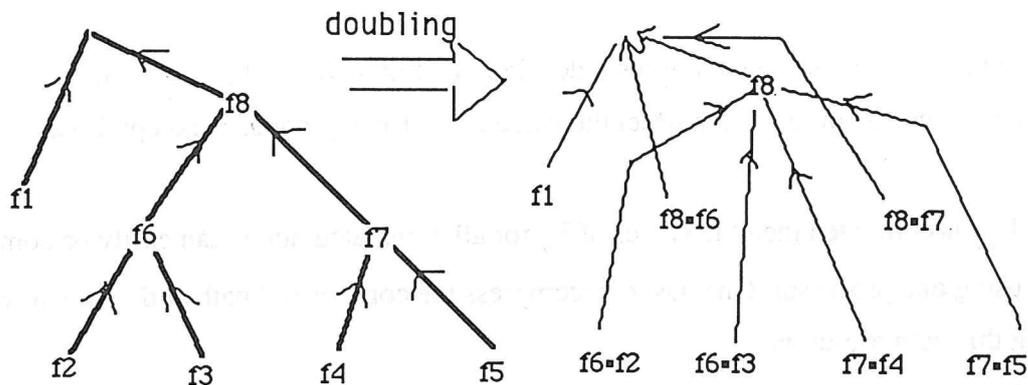


Fig. 4

Then execute the following.

```

repeat  $\log(n)$  times
  for each internal nonroot node  $v$  do in parallel
    begin  $F_v := F_{\text{father}(v)} \cdot F_v$ ;  $\text{father}(v) := \text{father}(\text{father}(v))$  end.

```

One step of this algorithm is illustrated in Fig.4. The above algorithm works in  $\log(n)$  time using  $n$  processors. In order to reduce the number of processors by a factor of  $\log n$  some preprocessing is required. The tree  $T$  is transformed to the reduced tree  $RT$  with  $n/\log n$  nodes and functions  $F_v$  are computed only for nodes of this tree using the above algorithm. Then the tree  $RT$  is expanded and the functions are computed for all nodes.

We say that a path from  $v_1$  to  $v_2$  (in a given tree  $T$ ) is reducible if each node on this path, except maybe  $v_1$  and  $v_2$ , has a son which is a leaf. (Formally, a single edge is also a reducible path, though there is no significant use of such a type of reducibility).

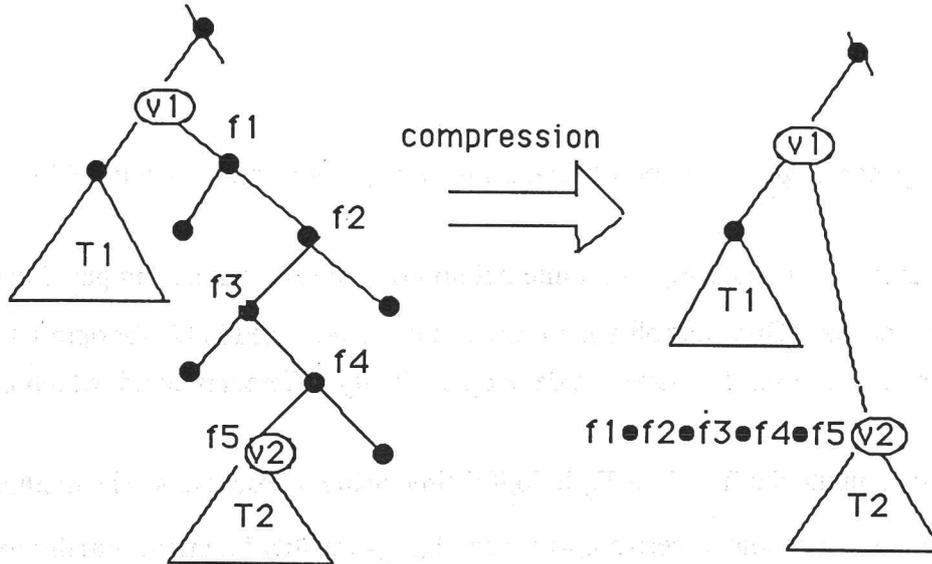


Fig.5

The reducible path  $p$  can be compressed into single edge, see Fig.5. We call this operation  $\text{compress}(p)$ . The compression does not affect the value of  $F_v$  for any nodes  $v$  except those eliminated.

If the value of  $F_{v_1}$  is computed then the values of  $F_v$  for all eliminated nodes can easily be computed in  $\log(n)$  time using one processor. One has to decompress the compressed path and compute values  $F_v$  going along this path top-down.

Our aim is to find  $n/\log(n)$  edge-disjoint reducible paths whose removal will reduce the tree by a factor of  $\log(n)$ . We use the approach of Bar-on and Vishkin. We refer the reader to [1], page 351 and assume a familiarity with the method and (especially) the claim related to subintervals induced by brackets chosen in step (3) of the Bar-on, Vishkin's algorithm.

At this moment we have the parse tree  $T$  without labels (with the functions  $D_v$  computed). Such a tree and the bracket structure of the input text are two different representation of the same object. The bracket structure will help to find a good decomposition into reducible paths.

We illustrate the method on the following example. Let

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

In the general case, partition the text into  $n/\log(n)$  segments of the length (approximately)  $\log n$ . For ease of presentation example let us disregard for a moment the true value of  $\log(n)$  and assume that the partition is as follows:

$$(((a)((a)((a)((a)(a)(a))))((a)(a))))((a)((a)(a)(a))))(((a)((a)((a)(a))))(a)(a))(a)(a)$$

Assign a processor to each segment. Each segment contains some matching pairs of brackets and some brackets whose matches are outside the segment. The processor (assigned to a given segment) finds which brackets have their matches inside the segment. Let us define the maximal matching pair in a given segment to be a pair of matching brackets, both within this segment, which is not enclosed by any pair with the same property. The subsegment whose endpoints are such brackets (including them) is called the maximal subsegment. Consider the second segment





Case 1.  $v_1=v_2=v$  (for example when  $v'=7$  in our tree)

The nodes  $v'$  and  $v$  are marked. The brackets corresponding to  $v'$  enclose the brackets corresponding to  $v$ .

We have the following situation if we disregard the symbol  $\bullet$  :

$$(\underset{v'}{(1(2(3((\dots(\underset{v}{\dots}))\dots))\dots))\dots))\dots)$$

All brackets between  $(\underset{v'}$  and  $(\underset{v}$  are left brackets and all brackets between  $)_{\underset{v'}$  and  $)_{\underset{v}}$  are right brackets. Moreover the bracket  $(_1$  matches the bracket  $)_1$ ,  $(_2$  matches  $)_2$ , and so on. It can be proved using the same argument as in [1], page 351. We refer the reader to a claim proved there.

Now look at two consecutively enclosed pairs of brackets, for example  $(_1)_1$  and  $(_2)_2$ . The node  $x$  corresponding to the first pair is the father of the node  $y$  corresponding to the second pair. Look at the other son  $z$  of the node  $x$ . We claim that this other son is a leaf. If we look at these pairs of brackets in the string  $w'$  then there are only two possible situations.  $(_1^\bullet(2\dots)_2)_1$  or  $(_1(2\dots)_2^\bullet)$ . The son  $z$  should correspond to  $\bullet$ . If it were not a leaf then it could be reduced to one leaf during the preprocessing of one segment, because all its leaves are in the same segment (they are between brackets lying in the same segment). Hence the path from  $v$  to  $v'$  is reducible.

Case 2.  $v_1 \neq v_2$  (for example when  $v'=4$  in our tree)

We have the following situation

$$(\underset{v'}{(1(2(3((\dots(\underset{k}{(v_1 \dots)_{v_1}(v_2 \dots)_{v_2}}\dots))\dots))\dots))\dots))\dots)$$

Now the node  $v''$  corresponding to  $(\underset{k}{\dots})_{\underset{k}}$  has both sons marked (because of marking corresponding brackets), hence it is a marked node (after our additional marking). It follows that the path from  $v'$  to  $v''$  is reducible using the same argument as in the previous case.

Take any nonroot marked node  $v$  and its path  $\text{path}(v)$  which ends at  $v'$  (a proper ancestor of  $v$ ). We have proved that if we go down from  $v'$  then at some moment we encounter a marked node  $x$  and the path from  $v'$  to  $x$  is reducible. However  $x$  must be equal to  $v$ , because there are no marked nodes between  $v$  and  $v'$  and  $v$  is not a leaf. Hence  $\text{path}(v)$  is reducible.

It is easy to see that there are only  $O(n/\log(n))$  marked brackets (constant number per segment), hence there are also only  $O(n/\log(n))$  marked nodes. There can be in fact more marked nodes than pairs of matching marked brackets (because of additionally added marked nodes). However it is of the same order. Hence the size of compressed tree is  $O(n/\log(n))$ . This completes the proof of the claim.

We have described at the beginning of the proof (of the theorem) an algorithm to compute all functions  $F$  in  $\log(m)$  time with  $m$  processors for a tree with  $m$  nodes. We compress our tree and

compute the functions  $F$  for the compressed tree  $RT$ . The size of  $RT$  is  $n/\log(n)$ . If we take  $m=n/\log(n)$  and the algorithm mentioned above, then we can compute the functions for  $RT$  in  $\log(n)$  time using  $n/\log(n)$  processors. Now we can assign a processor to  $\text{path}(v)$ , for each marked nonroot node, and we can easily decompress each such a path and compute functions  $F$  for reconstructed nodes in  $\log n$  time per one path using one processor. This requires time  $\log n$  and  $n/\log(n)$  processors. This completes the proof.

#### Remark

Our tree compression is a refinement of a compression used in [2]. It is especially useful when a function is to be computed for all nodes of the tree, because the decompression is easy. The technique from [2] is best suited for computing the value associated only with the root.

#### Acknowledgment

M.S.Paterson and A.M.Gibbons have made many helpful comments about this paper.

#### References

- [1] I.Bar-On, and U.Vishkin., Optimal parallel generation of a computation tree form. ACM Trans.on Progr.Lang.and Systems 7:2 (1985),348-357
- [2] A.Gibbons,W.Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. accepted for Found.of Software Techn.and Theoretical Comp.Science,december (1986), to appear in Lect.Notes in Comp.Science,Springer-Verlag.
- [3] Fortune,S.and Wyllie,J. Parallelism in random access machines, Proceedings of the 10th ACM Symp. Theory of Comp.(1978) 114-118 .
- [4] G.Kindervater,J.Lenstra. An introduction to parallelism in combinatorial optimization. Report OS-R8501, Centre for Math.and Comp.Science,Amsterdam (1984)
- [5] R.Mattheyses,and C.M.Fiduccia, Parsing Dyck languages on parallel machines. 20th Allerton Conference on Comm.Control and Computing (1982)
- [6] K.Mehlhorn. Bracket languages are recognizable in logarithmic space. Inf.Proc.Letters 5:6 (1976) 169-170
- [7] W. Ruzzo. On the complexity of general context free language parsing and recognition. Automata, languages and programming. Lect.Notes in Computer Science (1979),pp.489-499
- [8] W.Rytter. On the complexity of parallel parsing of general context-free languages. Accepted for Theoretical Computer Science
- [9] W.Rytter. On the recognition of context free languages. Computation Theory, Lect.Notes in Comp.Science 208,Springer Verlag (1985),pp.318-325
- [10] W.Rytter. An application of Mehlhorn algorithm for bracket languages to log n space recognition of input driven languages. Inf.Processing Letters,23:2 (1986) 81-84
- [11] W.Rytter. The complexity of two-way pushdown automata and recursive programs. in Combinatorial algorithms on words (ed.A.Apostolico,Z.Galil),Springer-Verlag (1985) 341-356
- [12] W.Rytter. Parallel time  $\log n$  recognition of unambiguous cfl's. Fund.of Computation Theory, Lect.Notes in Computer Science 199 (1985) 380-389,full version to appear in Inf. and Control