Research report 104

ON EFFICIENT PARALLEL COMPUTATIONS FOR SOME DYNAMIC PROGRAMMING PROBLEMS

Wojciech Rytter*

(RR104)

Abstract

A general method for parallelism of some dynamic programming algorithms on VLSI was presented in [6]. We present a general method for parallelisation for the same class of problems on more powerful parallel computers. The method is demonstrated on three typical dynamic programming problems: computing the optimal order of matrix multiplications, the optimal binary search tree and optimal triangulation of polygons (see[1,2]). For these problems the dynamic programming approach gives algorithms having a similar structure. They can be viewed as straight line programs of size $O(n^3)$. The general method of parallelisation of such programs descirbed by Valiant et al [16] then leads directly to algorithms working in $\log^2$ time with $O(n^9)$ processors. However we adopt an alternative approach and show that a special feature of dynamic programming problems can be used. They can be thought as generalized parsing problems: find a tree of the optimal decomposition of the problem into smaller subproblems. A parallel pebble game on trees [10,11] is used to decrease the number of processors and to simplify the structure of the algorithms. We show that the dynamic programming problems considered can be computed in $\log^2 n$ time using $n^6/\log(n)$ processors on a parallel random access machine without write conflicts (CREW P-RAM). The main operation is essentially matrix multiplication, which is easily implementable on parallel computers with a fixed interconnection network of processors (ultracomputers, in the sense of [15]). Hence the problems considered also can be computed in $\log^2 n$ time using n^6 processors on a perfect shuffle computer (PSC) or a cube connected computer (CCC). An extension of the algorithm from [14] for the recognition of context-free languages on PSC and CCC can be used. If the parallel random access machine with concurrent writes (CRCW P-RAM) is used then the minimum of m numbers can be determined in constant time (see [8]) and consequently the parallel time for the computation of dynamic programming problems can be reduced from $\log^2(n)$ to $\log(n)$. We investigate also the parallel computation of trees realising the optimal cost of dynamic programming problems.

* On leave from Institute of Informatics, Warsaw University,
Palac Kultury i Nauki 8 p, sk.poczt.1210, 00-901 Warszawa,
Poland

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

June 1987

The basic model of parallel computations considered in this paper is a parallel random access machine (P-RAM). Such a machine consists of a number of synchronously working processors (which are uniform cost RAM's) using a common memory.
The action of the parallel instruction:

for each x satisfying a given condition do in parallel instruction(x)

consists of assigning a processor to each x (if a specified condition holds) and executing instruction(x) for all such x simultaneously.

The model is called CREW P-RAM if no two processors can write simultaneously into the same location (however many processors can read at the same time from the same location). CREW stands for concurrent reads exclusive writes.
If we allow write conflicts (as well as read conflicts) then the model is known as CRCW P-RAM.
We consider also two parallel computers with a fixed interconnection network of processors: the perfect shuffle computer (PSC) and the cube connected computer (CCC). We refer the reader to [3] for definitions of PSC and CCC.

Many dynamic programming problems can be reduced to the computation of recurrences of the following type:

$$(*)\quad cost(i,j)=\min\{\, cost(i,k)+cost(k,j)+f(i,k,j) \;:\; i<k<j \,\},\ \text{for } 0\le i\le j<n,\ j-i\ge 2;$$
$$cost(i,i+1)=init(i),\ \text{for } i=0..n-1,$$

where the values of f(i,k,j) and init(i) are nonnegative integers known in advance.
The value of cost(0,n) and a tree T realising this value are to be found. We specify later what we mean by such a tree T when reformulating the problem as (**).

The recurrences (*) can be interpreted as follows: cost(i,j) is the cost of the problem with parameters (i,j). The problem with parameters (i,j) is decomposed into subproblems with parameters (i,k) and (k,j). The total cost is the sum of the costs of subproblems plus the additional cost of the decomposition. The value f(i,k,j) corresponds to the cost of the decomposition.

Example (minimum cost to evaluate the product of n matrices)
We consider the evaluation of the product of n matrices

$$M = M_1 \otimes M_2 \otimes \ldots \otimes M_n ,$$

where M_i is a matrix with r_{i-1} rows and r_i columns. Assume that the cost of multiplying a matrix with k rows and l columns by a matrix with l rows and j columns is k.l.j . The order in which the matrices are multiplied together can have some effect on the total cost of the evaluation. Let $m_{i,j}$ be the minimum cost of computing $M_{i+1} \otimes M_{i+2} \otimes \ldots \otimes M_j$. We have

$m_{i,i+1}=0$, for i=0...n-1;

$m_{i,j}$= MIN $\{m_{i,k}+m_{k,j}+r_i r_k r_j : i<k<j \}$, for j-i≥2.

Hence in this case init(i)=0 and $f(i,k,j)= r_i r_k r_j$. We can compute all products $r_{i-1} r_k r_j$ in O(1) time with n^3 processors and assume later that these values are precomputed constants.

Remark

One can easily convert the recurrences (*) into a straight-line program P (see[16]) with operations min and +. However such a program consists of $O(n^3)$ assignment statements. The size of P is $O(n^3)$ and the degree is O(n). If we want to have a distinct variable on the left side of each statement than we have to introduce variables $x_{i,k,j}$ corresponding to cost(i,k)+cost(k,j)+f(i,k,j). Using the method from [16] one can obtain directly a $\log^2(n)$ parallel time algorithm which uses $O(n^9)$ processors. However the structure of (*) is very special and we construct parallel algorithms for this specific type of recurrences which are much simpler than the algorithms derived by the very general method of [16]. Moreover the number of processors is reduced considerably. Our method also shows a close relationship between evaluation of expressions and the computation of dynamic programming problems. It is an application of a parallel pebble game introduced by the author [10].

Intuitively dynamic programming problems can be thought as instances of the following parenthesization problem: given a string $a_1 a_2 ... a_n$ of n objects, find a minimum (in a certain sense) parenthesization of the string. Then the optimal tree T is the tree corresponding to the optimal parenthesization and cost(i,j) is the minimum cost of the parenthesization of the substring $a_{i+1} ... a_j$. We now givea more formal definition.

Denote by S the set of all trees T with weighted nodes such that

(i) the nodes of T are pairs (i,j), 0≤i<j≤n;

(ii) if (i,j) is an internal node then its sons are of the form (i,k), (k,j), for i<k<j, and weight(i,j)=f(i,k,j); the weights are nonnegative numbers;

(iii) the leaves of T are (i,i+1), for 0≤i<n, and weight(i,i+1)=init(i).

Define the weight W(T) of a tree T as the sum of the weights of the nodes of T. Let

w(i,j) = min $\{W(T) : T \in S$, the root of T is $(i,j)\}$.

In this context, the tree T which realises the minimal weight also realises the minimum of cost(i,j). It is easy to see that w(i,j) = cost(i,j). Let S' be the set of trees from S whose root is (0,n). Now the dynamic programming problem related to the recurrences (*) can be reformulated as follows:

(**) : find the minimum weight of a tree $T \in S'$.

The crucial (though auxiliary) concept in parallel computations relating to such trees is that of a

partial tree, or a tree with a gap. This is a tree T from the set S rooted at some vertex (i,j) with one of its nodes (p,q) treated as a leaf. In other words it is a tree T with the subtree T' rooted at (p,q) deleted, except (p,q). T' can be treated as a gap (missing subtree), all nodes but the root of T' are missing. More formally, we say that the node (p,q) is the gap of T. For a given (i,j) and (p,q) we denote the set of such partial trees rooted in (i,j) with the gap (p,q) by PT(i,j,p,q).

The (partial) weight PW(T) of a partial tree $T \varepsilon PT(i,j,p,q)$ is the sum of the weights of all its nodes except the node (p,q). Let

$$pw(i,j,p,q)=MIN\{ PW(T):\ T \varepsilon PT(i,j,p,q) \}.$$

Observe that pw(i,j,i,j)=0.
Let T be a tree rooted at (i,j) ,where the sons of (i,j) are (i,k), (k,j). Let T1 be the tree rooted at (k,j) and T' be the partial tree rooted at (i,j) with the node (i,k) treated as a leaf ((i,k) is the gap). Then
(1') PW(T')= f(i,k,j)+ W(T1).
This implies the following equality:
(1a) pw(i,j,i,k)= f(i,j,k)+w(k,j), where (k,j) is the right son of (i,j) in the tree realising the minimum of cost(i,j).
Similarly one can derive the equality
(1b) pw(i,j,k,j)= f(i,j,k)+w(i,k), where (i,k) is the left son of (i,j) in the tree realising the minimum of cost(i,j).

Let T be a partial tree with the root (i,j) and the gap (p,q) and let (r,s) be an intermediate node on the path from (i,j) to (p,q). Denote by T1 the subtree of T rooted at (i,j) with the gap (r,s), and by T2 the subtree of T rooted at (r,s) with the gap (p,q). Then
(2') PW(T) = PW(T1) + PW(T2) and the following equality follows:

(2) $pw(i,j,p,q)= MIN\{ pw(i,j,r,s)+pw(r,s,p,q) :\ i \le r \le p \text{ and } q \le s \le j \}$

If T is a tree rooted at (i,j), T1 is a subtree of T rooted at an internal vertex $(p,q) \neq (i,j)$ and T2 is a partial subtree rooted at (i,j) with gap (p,q) then
(3') W(T) = W(T1) + PW(T2).
This implies the equality:

(3) $w(i,j)=MIN\{ pw(i,j,p,q)+w(p,q) :\ i \le p < q \le j,\ (p,q) \neq (i,j) \}.$

We introduce the auxiliary arrays w'(i,j) and pw'(i,j,p,q). At the end of the algorithm we want w'=w and pw'=pw. Initially all entries of introduced arrays w' and pw' contain the value $+\infty$, except entries w'(i,i+1)=init(i). Then in the course of the algorithm some of the values will decrease.
We introduce also three parallel operations activate1, square1 and pebble1 which correspond to the equalities, respectively, (1a) and (1b), (2), (3).

activate1: for each $0 \le i < k < j \le n$ do in parallel

$pw'(i,j,i,k) := \text{MIN}\{ pw'(i,j,i,k), f(i,j,k)+w'(k,j)\}$;

$pw'(i,j,k,j) := \text{MIN}\{ pw'(i,j,k,j), f(i,j,k)+w'(i,k) \}$;

square1: for each $0 \le i < p < q < j \le n$, $j-i \ge 2$ do in parallel

$pw'(i,j,p,q) := \text{MIN}\{ pw'(i,j,r,s)+pw'(r,s,p,q) : i \le r \le p,\ q \le s \le j \}\}$

pebble1: for each $0 \le i \le p < q \le j \le n$, $j-i \ge 2$ do in parallel

$w'(i,j) := \text{MIN}\{ pw'(i,j,p,q)+w'(p,q) : i \le p < q \le j\}$.

The whole algorithm is now very short.

Algorithm Evaluate;
repeat $\log_2 n$ times
begin
activate1; square1; square1; pebble1
end.

Theorem 1

(a) After termination of the algorithm Evaluate we have $w'(i,j)=w(i,j)$ for each $0 \le i < j \le n$.
(b) The recurrences (*) can be computed in $\log^2 n$ time using $n^6/\log(n)$ processors on a CREW P-RAM.
(c) The recurrences (*) can be computed in log(n) time using a polynomial number of processors on a CRCW P-RAM.
(d) The recurrences (*) can be computed in $\log^2(n)$ time using $O(n^6)$ processors on a PSC or on a CCC.

Before starting the proof we describe a parallel pebble game on binary trees. The game was first introduced by the author in [10] for the parallel evaluation of recursive programs with independent calls (which programs for evaluating algebraic expressions are a special case) and later used in an optimal parallel algorithm for the dynamic evaluation of expressions (see [4]). The concept of parallel pebbling was also used by the author in log(n) time recognition of unambiguous context free languages [13]. The parallel pebble game is very similar to the tree contraction concept of Miller and Reif [9], though these were invented independently of each other. Within the game each node v of the tree has associated with it a similar node denoted by cond(v). At the outset of the game cond(v) = v, for all v. During the game the pairs (v,cond(v)) can be thought of as additional edges. Another notion we shall require is that of 'pebbling' a node. We have an unlimited number of pebbles. A pebble is placed on a node is never subsequently removed. At the outset of the game only the leaves of the tree are pebbled.

We say that a node v is 'active' if and only if cond(v) ≠ v.
The three operations activate, square and pebble are components of a 'move' within the game and are defined as follows:

activate:
for each nonleaf node v do in parallel
if v is not active and precisely one of its sons is pebbled then cond(v) becomes the other son
if v is not active and both sons are pebbled then cond(v) becomes one of the sons arbitrarily

square:
for each node v do in parallel cond(v):=cond(cond(v));

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

Now we define one (composite) move of the pebbling game to be the sequence of individual operations:

(activate; square; square; pebble)

in that order. Assume for simplicity that n is a power of two. Then the following fact provides a key result.

Fact

Let T be a binary tree with n leaves. If initially only the leaves are pebbled then after $\log_2(n)$ composite moves of the pebbling game the root of T becomes pebbled.

The fact was proved in [11]. We present a main idea of the proof. We define a modified pebbling move consisting of the sequence:

(pebble; activate; square; square)

It is enough to prove that after $(\log_2(n)+1)$ such moves the root will be pebbled. This is because if a node is pebbled after (k+1) of these leaves moves then it will be pebbled after k of the original moves. The first pebble operation and the last individual operations activate, square, square are redundant in this context. Let size(x) denote the number of leaves of T_x, the binary (sub)tree rooted at x. By size (x/y) we mean (size(x) - size(y)). We number the modified moves from 0 to log(n). The followng claim can be proved.

Claim

After the k-th modified move the following invariants hold for each node x of the tree:

(I1) if $\text{size}(x) \leq 2^k$ then x is pebbled

(I2) $(\text{size}(x/\text{cond}(x)) \geq 2^k)$ or (no son of cond(x) is pebbled) or (cond(x) is a leaf).

Remark

In [10] one composite move was defined as a sequence: (activate; square; pebble). It was proved there that O(log(n)) such moves are sufficient to pebble the root. Hence alternative algorithms for dynamic programming problems follow by using the sequences of this type of composite moves. In this case the constant coefficient before $\log_2 n$ is bigger than two, while in the case of composite moves with two operations square the coefficient is the smallest possible (it equals one).

Proof of theorem 1.

(a)

It is easy to see that $w'(i,j) \geq w(i,j)$ and $pw'(i,j,p,q) \geq pw(i,j,p,q)$ for every i,j,p,q in the course of the algorithm. It is enough to show that at some stage in the algorithm the equalities can be obtained.The essential point is to prove that $\log_2(n)$ repetitions are sufficient. We do this using a relationship between the operations activate1, square1 and pebble1 and the operations activate, square, pebble as defined for the parallel pebble game .

Consider a pair (i,j), $0 \leq i < j-1 < n$. Consider a particular tree $T \in S$ (one of possibly many) with minimal weight and with the root (i,j). The weight of this tree equals w(i,j) and for each node (p,q) of T the weight of the subtree of T rooted at (p,q) equals w(p,q). Moreover the weight PW(T') of the partial tree T' which is a subtree of T with root (i,j) and gap (p,q) equals pw(i,j,p,q). Hence as far as nodes of T are only concerned the weights pw' and w' reach their minimal values in computations involving only nodes of T. Therefore in considering the final value of w'(i,j) we can ignore all pairs (k,l) which are not the nodes of T.

Now we play the parallel pebble game on T in the course of the algorithm Evaluate . We consider an extended version Evaluate1 of the algorithm Evaluate. Assume that initially all the leaves of T are pebbled.

Algorithm Evaluate1;
repeat $\log_2 n$ times
begin
activate; activate1; square; square1; square; square1; pebble; pebble1
end.

It is easy to see that the following two invariants hold after each of the operations activate1; square1; pebble1:

If (p,q) is pebbled then w(i,j)=w'(p,q)) and
if cond(p,q)=(r,s) then pw(p,q,r,s)=pw'(p,q,r,s),

for every two nodes (p,q) and (r,s) of the tree T.

(In other words if the node is pebbled then its weight w' is reaching its minimal value. Similarly for pairs of nodes related through the function cond and partial weights pw'.)

Initially only the leaves of T are pebbled, but all of them are of the form (i,i+1) and their values

w'(i,i+1)=w(i,i+1) are correctly computed and set to init(i). Then the equalities (1'), (2'), (3') and (1a), (1b), (2) and (3) can be used to show that in the course of the algorithm Evaluate1 the invariants are preserved.

Hence in the moment of pebbling the node (i,j) the value of w'(i,j) is correctly computed. However this node will be ultimately pebbled because of the fact about the parallel pebble game. Observe now that the pebble game performed on the tree T can be ignored and the operations activate, square and pebble removed; they are introduced only to show the correcteness. Then algorithm Evaluate1 becomes our initial algorithm Evaluate. Hence w'(i,j) is correctly computed in the algorithm Evaluate. The same argument applies to ever pair (i,j) by taking a suitable optimal tree T with the root (i,j). This completes the proof of point (a).

(b)
The biggest number of processors is required in operation square1. For every 4-tuple (i,j,p,q) we have to compute a minimum of $O(n^2)$ values. This can be done in log(n) time using $n^2/\log(n)$ processors for a fixed 4-tuple, or in the same time using $n^6/\log n$ processors for all 4-tuples simultaneously.

(c)
If we have concurrent writes then a minimum of n^2 values can be computed in O(1) time using $O(n^4)$ processors, see [8]. Each of the operations activate1, square1 and pebble1 can be performed in O(1) time with a polynomial number of processors. This proves point (c).

(d)
The operation square1 can be seen as a matrix multiplication with elements from a semiring, where the rows and columns correspond to pairs (i,j). Here the size of the matrix is $O(n^4)$. The operations involved are minimum and addition. Similarly operations activate1 and pebble1 can be implemented as matrix operations. The implementation is very technical. The same method as used in [14] for the recognition of context free languages on a PSC or on a CCC can be used. No new ideas are needed. We refer the reader to [14] and [3].
If the matrices have M rows and M columns then matrix multiplication can be done in log M time using M^3 processors on a PSC or CCC computer, see [3]. There are log(n) iterations. Therefore the whole algorithm can be implemented on a PSC or on a CCC in $\log^2 n$ time using n^6 processors. This completes the proof of the theorem.

Corollary 1.1
The minimum cost of the evaluation of the product of n matrices can be computed in $\log^2 n$ time using $n^6/\log(n)$ processors.

Our second example of the dynamic programming problem is the optimal triangulation of the polygon. We are given the nodes of a polygon and a distance between each pair of nodes. The

problem is to select a set of diagonals (lines between nonadjacent nodes) such that no two diagonals cross each other, and the entire polygon is divided into triangles. The cost is the minimum total length of the diagonals.

Let the polygon be given by its nodes (in clockwise order) $v_0,v_1,...,v_n$. The subproblem with parameters (i,j) is the computation of the minimal triangulation of the polygon given by nodes $v_i,v_{i+1},...,v_j$. Let $m_{i,j}$ be the cost of this subproblem. The equations similar to (*) in the previous example can be easily found to compute $m_{i,j}$. We refer the reader to [1] for details. Applying Theorem 1 we obtain

Corollary 1.2

The triangulation problem can be solved in $\log^2 n$ time using $O(n^6/\log(n))$ processors.

Our third example of the dynamic programming problem is the computation of the optimal binary search trees. Let K_1, K_2, ..., K_n be some keys given in an increasing order. Let p_i be the frequency of the access to the key K_i. We want to construct a binary tree T with the leaves K_1, K_2, ..., K_n . The cost of tree cost(T) is the sum of $l_i p_i$, for $1 \le i \le n$, where l_i is the length of the path from the root to K_i.

For $0 \le j < j \le n$ let $T_{i,j}$ be the minimum-cost tree for the subsequence of keys K_{i+1}, K_{i+2},...K_j and $m_{i,j}$ be the cost of this tree. Now we can write an equation similar to (*) for $m_{i,j}$, see [2]. Applying theorem 1 again we obtain.

Corollary 1.3

The cost of the optimal binary search tree can be computed in $\log^2 n$ time using $O(n^6/\log(n))$ processors.

Observe that in the last three examples not only is the minimum cost of interest: also so is an optimal tree. In the first example such a tree describes the optimal order of matrix multiplication (or bracket structure imposed), in the second example it corresponds to the structure of diagonalisation. The problem of recognizing context-free languages can be also formulated as a dynamic programming problem, we have to compute the boolean value which says if the string is generated by a given grammar or not. The parsing problem for context-free languages is to find a parse tree if one exists. The computation of optimal trees from the last three examples can be thought as a generalized parsing problem. It was shown in [12] that if we have computed the parsing matrix (see [12]) then the parsing tree can be found using an efficient parallel algorithm. The matrix cost(i,j) plays the same role in dynamic programming problems as the parsing matrix in the case of context free languages. We refer the reader to [12] for details. The next theorem follows by using the same method as that used in [12] for the computation of parse trees from parsing matrices.

Theorem 2

The optimal tree T realising the minimal value of cost(0,n) in the recurrence (*) can be computed in $\log^2 n$ time using n^6 processors on a CREW P-RAM.

The theorem implies (for example) that the optimal binary search tree, not only the cost of such tree, can be efficiently found using a parallel algorithm. The same applies to the other two dynamic programming problems considered in this paper.

Remark

In the case of context-free languages it was shown in [13] that if we know in advance that the parsing tree is unique (if it exists) then log(n) parallel time algorithms on a CREW P-RAM are possible. Does the same apply generally to the dynamic programming problems if we assume that the optimal tree is unique?

Parallel algorithms for dynamic programming problems on the P-RAM were also investigated in [10] in the framework of path systems and recursive programs. The algorithms presented in this paper can be generalized and can also be presented using the terminology of path systems. A path system G is given by a 4-tuple (N,Ter,s,R), where N is a finite set of elements, Ter is a subset of N (its elements are called terminal elements), $s \in N$ is a goal element and R is a ternary relation on N. (x,y,z) is in R iff z is a possible father of x and y. We associate nonnegative weights with elements of Ter and triples from R. The derivation tree for the goal element s is a tree T whose root is s, whose leaves are in T and whenever x,y are sons of an element z in T then (x,y,z) is in R. The weight of an internal node z whose sons are x,y is the weight of the triple (x,y,z). The weight of T is defined to be the sum of all weights of leaves plus the weights of internal nodes. Define the cost c(G) of the system G to be the minimal weight of a derivation tree for s. The cost equals $+\infty$ if there is no such tree.

The problems of computing the dynamic programming problems and of computing the weight of minimal tree in the sense of (**) are special cases of a more general problem of computing c(G) for a given weighted path system G. In these problems the elements of N are pairs (i,j), the terminal elements are pairs (i,i+1) and (x,y,z) is in R iff x=(i,k), y=(k,j) and z=(i,j) for some $0 \le i < k < j \le n$. The weight of such a triple (x,y,z) is f(i,k,j). The goal element is (0,n).

The algorithm Evaluate can be easily extended to compute c(G) for any weighted path system such that the sizes of the derivation trees for the goal element s are bounded by a polynomial.

However the problem of computing c(G) for general path systems is P-complete, which follows easily from the fact that the problem of solvability (verification if the goal node can be derived from terminal elements) for path systems is P-complete.

One can consider proof systems instead of path systems . The terminal nodes are now axioms and the relation R represents rules which are weighted. Then the problem consists of finding for a given theorem a proof whose cost is minimal.

Acknowledgment

The author thanks W.M. Beynon for linguistic help.

Bibiography.

[1] A.Aho, J.Hopcroft and J.Ullman. Data structures and algorithms. Addison-Wesley (1983)

[2] A.Aho, J.Hopcroft and J.Ullman. The design and analysis of computer algorithms. Addison-Wesley (1974)

[3] E.Dekel,D.Nassimi,S.Sahni. Parallel matrix and graph algorithms. SIAM Journal on Computing 10:4 (1981)

[4] A.Gibbons, W.Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. in Found.of Software Techn.and Theoretical Comp.Science, Lecture Notes in Computer Science, Springer Verlag (1986).

[5] A.Gibbons, W.Rytter. Efficient parallel algorithms: an introduction. The book, Cambridge University Press, to appear (1987).

[6] L.Guibas, Kung H, Thompson C. Direct VLSI implementation of combinatorial algorithms. Caltech Conference on VLSI, (1979) 509-525

[7] G.Kindervater and J.Lenstra,An introduction to parallelism in combinatorial optimization. Report OS-R8501,Centre for Mathematics and Computer Science, Amsterdam (1985)

[8] L.Kucera. Parallel computation and conflicts in memory access. Inf.Proc.Letters 14:2 (1982) 93-96

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

[10] W.Rytter, 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 (June, 1985) (the conference took place in June 1984)

[11] W.Rytter, Remarks on pebble games on graphs. Presented at the conference 'Combinatorial analysis and its applications' (September, 1985) (ed.M.Syslo), the proceedings published in Zastosowania Matematyki (1987).

[12] W.Rytter.On the complexity of parallel parsing of general context-free languages. Theoretical Computer Science (1987)

[13] W.Rytter. Parallel time O(log(n)) recognition of unambiguous context free languages. Information and Computation 73:1 (1987) 75-86

[14] W.Rytter. On the recognition of context free languages. Computation Theorey (ed.A.Skowron), Lect.Notes in Computer Science 208, Springer-Verlag (1985) 318-325

[15] J.Schwartz. Ultracomputers. ACM Trans. on Programming Languages and Systems 2:4 (1980) 454-521

[16] L.Valiant,S.Skyum,S.Berkowitz, C.Rackoff. Fast parallel computation of polynomials using few processors. SIAM J.Comp. 12:4 (1983) 641-644