COLLOQUIA MATHEMATICA SOCIETATIS JÁNOS BOLYAI
26. Mathematical Logic in Computer Science,
Salgótarján (HUNGARY), 1978.

LOGIC AND COMPLEXITY OF SYNCHRONOUS PARALLEL COMPUTATIONS

S. Radziszowski

ABSTRACT

We investigate a certain model of synchronous parallelism. Syntax, semantics and complexity of programs within it are defined. We consider algorithmic properties of synchronous parallel programs in connection with sequential programs with arrays. The complexity theorem states that the class PP-time (polynomial-time bounded parallel languages) is equal to P-space (languages requiring polynomial amount of memory).

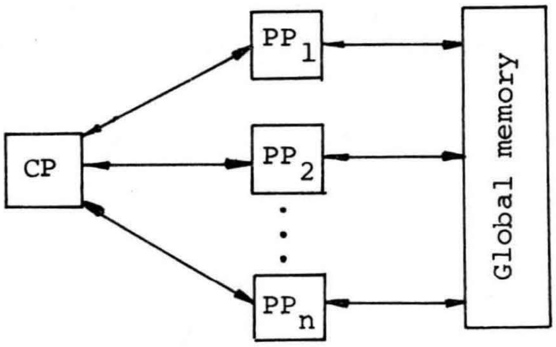
INTRODUCTION

In the recent years many papers appeared investigating different kinds of models for synchronous parallel computations. In general, they are divided into two groups, which are dealing with:

1. Creation of new formal algerbraic models for parallel computations, such as vector machines (Pratt, Stockmeyer [9]), alternating Turing machines (Chandra, Stockmeyer [2]), M-Ram, C-Ram (Simon [6]), conglomerates (Goldschlager [5]) and others.
2. Practical parallel programming in languages not formally defined, with intuitive semantics. Even examples of algorithms from the first group of papers are often written in such languages.

Our goal is to present a certain very natural language with the complete definition of syntax and semantics. On the basis of this language we investigate some algorithmic properties of parallelism, the complexity of programs written in it and we give some examples of algorithms. Such an approach might turn out to be directly applicable to future parallel computers.

The general idea of our model is as follows:



CP - the control processor stores the text of a program and synchronizes the actions of other processors PP_1, PP_2,...,PP_n,...

PP_i - $i>0$, there is an unbounded number of processors indexed by natural numbers, acting in parallel, all of them using one global memory M. In the language syntax there does not exist any notion of processor. We have only to specify which instructions should be executed in parallel.

The whole structure acts in sequential and parallel steps. A sequential step is the execution of one standard statement at a given moment. One parallel step has two stages. First the number of active processors and their allocation are computed by the control processor CP. After that every active processor PP_i performs one sequential instruction (which can be a sequential program in general). This model is stronger than SIMDG - single instruction stream, multiple data stream, global memory (Flynn [4]) because in our model different processors PP_i can perform in one step different programs, contrary to SIMDG, where two parallel processors must be executing the same instruction if they are both active in a given time. However, in our model the number of different instructions executed in parallel is syntactically bounded by the text of the program stored in the control processor CP.

By synchronous parallel program we shall intuitively mean the program whose computation is deterministic and all parallel instructions are mutually separated. All processors act in tacts, that means there are syntactically defined points in the program, where all processors are forced to synchronize their actions.

We use only synchronous parallel computations, because we are interested in fast algorithms solving particular problems. Up to now asynchronous model of parallelism is mostly used for on-line computations, operating system and so on. There exist also some numerical asynchronous parallel algorithms (Kung [8]).

Asynchronous parallel programming does not apply to particular problem solving, such as language recognition. It seems that by removing synchronization we canot essentially improve the complexity of algorithm, for the worst case complexity of asynchronous algorithm will always remain not smaller than the complexity of the corresponding synchronized version. At most, an average running time of the program can be decreased.

I. PARALLEL PROGRAMS

Let R be a relational system:

$$R = \langle A, \{f_l\}, \{r_j\} \rangle,$$

where the set of natural numbers forms a subset of A and f_i, r_j are the functors and predicates of R. Let us denote by FS_R the set of sequential programs in R, i.e. containing substitutions and closed under composition (block statement), condition (if statement) and iteration (while statement), (see Banachowski et al [1]).

The set of variables V consists of infinite sets V_b, V_s, V_n, V_i:

$$V = V_b \cup V_s \cup V_n \cup V_i$$

where all of them are pairwise disjoint and

V_b - boolean variables,

V_s - standard individual variables valuated into the set A,

V_i - index variables valuated into the set of natural numbers.

Let $\overline{V}_n$ be an infinite set of names, then:

$V_n = \{x(k): x \in \overline{V}_n$ and $k \in N\}$ is the set of simple indexed variables.

The terms and the formulas are built by induction on the basis of variables from the set V and functors and predicates from R.

If the valuation $v: V_b \cup V_s \cup V_i \cup V_n \to A$ is given, then we shall use an extended valuation v for complex indexed variables of the form $x(\tau)$, where τ is a term. In this case we define:

$$x(\tau)\ (v) = \begin{cases} x(\tau(v)) & \text{if } \tau(v) \text{ is a natural number} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Let us denote by $\overline{FS}_R$ an extended set of sequential programs, where the set of variables is equal to $\overline{V} = V_b \cup V_s \cup V_i \cup \{$simple and complex indexed variables$\}$.

Definition:

Parallel instruction is the program of the form:

cobegin $(I_1 \Box \rho_1), \ldots, (I_r \Box \rho_r)$ *coend*

where:

1. ρ_j is the relation programmable in R and it is writen in the form $K\alpha$ for some program $K \in \overline{FS}_R$ and an open formula α, for $j=1,\ldots,r$. (cf. [1]).
2. I_j for $j=1,\ldots,r$ is a sequential program from $\overline{FS}_R$.

3. For all $j=1,\dots,r$ the set of free index variables in I_j and ρ_j is the same.
4. For all $j=1,\dots,r$ any index variable in I_j can not occur as a left side of substitution. (This restriction is implied by semantics, because ρ_j will assign those variables on which program I_j will be executed in parallel)[1]. □

In some cases, which are not involving any confusion, we will use the simplified notation for parallel instructions, for instance:

a) *cobegin* $I \square \rho$ *coend*	if $r = 1$,
b) *cobegin* I, J *coend*	if I, J does not contain any index variable .

Definition:

The set of *parallel programs* PP is the smallest set satisfying the following conditions:

1. $\overline{FS}_R$ is included in PP;
2. Parallel instruction is a parallel program;
3. PP is closed under composition, condition and iteration on the basis of programs from $\overline{FS}_R$. □

Before the definition of semantics let us give an example of program sorting n different elements given in array B[1:n] . Our example is a somewhat improved

1) This condition is to avoid the variable conflict as in

```
begin  k:=2;  i:=2;
     cobegin  X(k):=4;  X(i):=0 coend
end
```

version of the algorithm from Goldschlager [5]. In the formalism of PP-programs we can restrict the range of an index j in *while* statement I2 to the interval $1 \leq j \leq n/2^k$.

Example 1

B[1],...,B[n] - elements from the ordered set A to be sorted

$R = \langle A \cup N, \leq_A, \text{arithmetic in } N \rangle$.

For the sake of convenience we shall use multiindexing of variables, i.e. a (i,j) instead of a (number of the pair (i,j)).

Program K:

```
begin comment perform all comparisons;
 I1: cobegin if B[j]≥B[i] then less(i,j):=1 else
                               less(i,j):=0□
                               1≤i,j≤n coend;
     comment compute the number of inpute less than
             or equal to input B[i];
 I2: K:=1; while k≤log₂n do
           begin
            cobegin less(i,j):=less(i,2j-1) +
            less (i,2j) □1≤i≤n∧1≤j≤n/2^k coend;
             k:=k+1
           end;
     comment re-arrange the input numbers;
 I3: cobegin B[less(i,1)]:= B[1]□ 1≤i≤n  coend
end;
```

Our program K sorts elements B[1],...,B[n] in time O(logn), assuming n is a power of 2.

Let K be a parallel program from PP and v an initial valuation into the set A. We shall define output valuation $v' = K_R(v)$.

1. $K \in \overline{FS}_R$, K is a sequential program, by standard inductive way we put $v' = K_R(v)$. (cf.[1]).
2. $K =$ *cobegin* $(I_1 \square \rho_1),\ldots,(I_r \square \rho_r)$ *coend*
   Let S_j be the set of all free variables from ρ_j.
   Denote:

   $T_j = \{(n_1,\ldots,n_{k_j}):\rho_j(n_1,\ldots,n_{k_j})(v) = 1\}$ for

   $j=1,\ldots$ The set T_j is the set of all sequence of index variables satisfying ρ_j. For each such sequence $n_1,\ldots,n_k$ a separate processor will execute program I_j, assuming it will not lead to the conflict. In order to omit conflicts we have to force the actions of all processors to be independent inside the parallel instruction. Formally, if we define t^j_ξ for $\xi \in T_j$:

   $t^j_\xi =$ {the set of all variables occuring in I_j as a left side of substitution, while the initial valuation for program I_j is given by v changed by ξ on variables from S_j (denote it by v^j_ξ)}

   then the resulting valuation v' will be defined if:

   a) all sets T_j, $j=1,\ldots,r$, are finite
   b) all sets t^j_ξ, for $j=1,\ldots,r$ and ξ ranging over T_j, are pairwise disjoint

   and v' is given by:

$$v' = K(v) = w \cup \bigcup_{j,\xi} I_j(v_\xi^j) \qquad (*)$$

where w is equal to the valuation v restricted to the set $V \setminus \bigcup_j S_j \setminus \bigcup_{j,\xi} t_\xi^j$. The formula (*) describes independent actions of $\sum_j |T_j|$ processors, which are allowed to change the valuation in the separate parts of memory t_ξ^j, but they have the possibility to read all variables.

Following conditions a), b) v' is correctly defined. Note, that v' can be undefined on the part of set $\bigcup_j S_j$.

3. if $K = K_1;K_2$ then $K_R(v) = K_{2R}(K_{1R}(v))$

   if K = *if* α *then* K_1 *else* K_2 then:

$$K_R(v) = \begin{cases} K_{1R}(v) & \text{if } \alpha_R(v) = 1 \\ K_{2R}(v) & \text{if } \alpha_R(v) = 0 \end{cases}$$

   if K = *while* α *do* K_1 then:

$$K_R(v) = \begin{cases} v & \text{if } \alpha_R(v) = 0 \\ K_R(K_{1R}(v)) & \text{if } \alpha_R(v) = 1 \end{cases}$$

Note that the output valuation v' can become undefined in three cases:

i) by the infinite loop in the *while* statement as in sequential programs

ii) by variable conflict as in 2b) of semantics definition, i.e. instruction of the form:

cobegin x:=a; x:=b *coend*

iii) by the requirement of an infinite number of processors (point 2a) of semantics definition).

Let us notice also that the conflict of the form:

cobegin x:=b; y:=x *coend*

is solved in the definition. Following the remark 4b) from the syntax definition appropriate sets will be like this:

$$T_1 = T_2 = \{\emptyset\}, \qquad S_1 = S_2 = \{\emptyset\}$$

$$v^1 = v^2 = v \ , \quad t^1 = \{x\}, \quad t^2 = \{y\}, \quad t^1 \cap t^2 = \emptyset$$

$$v' = (v \text{ restricted to } V - \{x,y\}) \cup [x/b](v^1) \cup [y/x](v^2)$$

The result is equivalent to the following sequential program:

```
begin  y1:=y; x1:=x;
       x1:=b; y1:=x;
       y:=y1; x:=x1  end
```

In a real computer acting in parallel the computation can be performed in the following way:

All sequential statements outside the parallel instructions are executed in a standard way. The general assumption about hardware is that all processors have access to the whole global memory. Each variable can be read many processors at the same moment, but only one processor can change the value of the variable at a given time. Before each parallel instruction the control processor computes the set of all sequences of index values satisfying ρ_j, whose cardinality gives the number of required processors. This can be done in parallel by the operating system. Control processor does not have to check whether the number of processors is finite. CP can simply print out the computated number or inform that its capacity is too small to activate all desired processors.

For every sequence satisfying ρ_j a new processor is activated which executes the program I_j in its local memory. The results are copied into the global memory. This is described in the semantics definition. The variable conflicts as in 2b) can be checked in running time by special marking of changed variables inside parallel instruction. The execution of parallel instruction is terminated when all processors have executed their programs.

III. ALGORITHMIC PROPERTIES OF PARALLEL PROGRAMS

Let $K \in PP$ be a parallel program, v - a valuation of its variables and α a formula. By $K\alpha$ we shall mean the formula with the following definition of valuation:

$$(K\alpha)_R(v) \overset{df}{=} \alpha_R(K_R(v))$$

In the same manner as in sequential case we would like to prove partial and total correctness of a program K in the structure R with respect to the formulas α and β, i.e. to prove the following formulas:

$(K1 \wedge \alpha) \rightarrow K\beta$	- partial correctness
$\alpha \rightarrow K\beta$	- total correctness

In the last two chapters of the paper we have to make some restrictions on the form of the programmable relation ρ_j. If we allow ρ_j to be an arbitrary formula $K\alpha$, then even the problem of finiteness ρ_j could become undecidable. In order to obtain effectiveness and complexity theorem we assume that ρ_j is given by a system of linear inequalities (with respect to index variables), where coefficients are arithmetic terms over standard variables.

In this case there exist fast algorithms checking satisfiability of ρ_j and computing the set of solutions for ρ_j (the special case of linear programming). On the other hand this restriction seems to be reasonable, because the structure of parallelism still remains powerful enough in practice.

THEOREM 1

There exists an effective semantically equivalent translation between the parallel and sequential programs with arrays.

Proof:

$\Leftarrow$ every sequential program is a parallel one. The cell of array can be treated as a single indexed variable. In such a transformed program, that is equivalent to the initial one, there are no simultaneous statements.

$\Rightarrow$ It is enough to give an efficient semantically equivalent transformation of an arbitrary parallel program to the sequential one with arrays. It is obviously sufficient to do this for a parallel instruction. One parallel instruction can be transformed as follows:

1. For every ρ_j, $j=1,\dots,r$ compute the set T_j of sequences satisfying ρ_j. If T_j is not finite (in the case of system of linear inequalities the problem is decidable) then stop without result.
2. Following the semantics definition check if there exists unavoidable variable conflict (point ii)). If yes then stop with undefined result.

3. Treat indexed variables with the same name as an array. Working on local memory for every sequence from $\bigcup_j T_j$ execute program I_j (sequentially).
4. Copy results into the global memory.

As an immediate consequence we have the following:

COROLLARY

The problem of partial (total) correctness of parallel program is equivalent to the problem of partial (total) correctness of sequential programs with arrays.

The transformation in the proof of the theorem gives the equivalent program since we have followed the definition of semantics. Our program is a sequential one with arrays. For the original study of algorithmic properties of programs with arrays see Dańko [3].

The correctness proofs of parallel programs can be made in particular case much simpler than in general by combining the transformation to the sequential program with arrays and proving its correctness. If one parallel instruction is not very complicated we can treat it as a single instruction and we apply standard methods to the whole program.

For instance, let us prove the total correctness of program K (Example 1) in natural numbers with respect to the formulas:

input formula α: $n\geq 1\wedge B[i]\in N\wedge(i \neq j \rightarrow B[i] \neq B[j])$
output formula β: $i, j\in N\wedge(i<j\rightarrow B[i]<B[j])$

We would like to prove that:

$$\alpha \rightarrow K\beta$$

is a valid formula in the natural numbers. The program K has the form:

$$\begin{tikzcd}
I_1 \arrow[r] & k \le \log_2 n \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"'] & I_2 \arrow[l, bend left] \\
& I_3 \arrow[d] & \\
& \text{Stop} &
\end{tikzcd}$$

After executing I_1, k is equal to 1 and less(i,j) codes the relation $B[i] \le B[j]$. The program halts, because every step of loop execution increases k and hence the loop will operate exactly $\log_2 n$ times. Then it is sufficient to show that the following formula γ is an invariant of the loop statement I_2:

$$\gamma = \bigwedge_{1\le i\le n} \Big(\sum_{1\le j\le n/2^k} \mathrm{less}(i,j) = \text{the number of elements smaller than or equal to } B[i] \text{ in the array} \Big)$$

It can be proved by simple induction. For $k = \log_2 n$ less [i,j] gives the number of elements smaller than B[i] and hence instruction I_3 leads to the correct result.

Such parallel programs can be used as a device for recognizing languages, by setting x_1:=input word at the start of the computation and leading out the result at the distinguished boolean variable.

IV. COMPLEXITY OF PARALLEL COMPUTATIONS

The main theorem of this section is a generalization of results from [2, 9].

The complexity of program from FS_R is measured by the number of instructions performed during the computation (Radziszowski [10]). The length of parallel instruction is the maximum of length of sequential computations together with the cost of relation ρ. (The cost of ρ depends on the way of compilation, but there exist fast algorithms for computing ρ. We do not specify these algorithms, because we are interested in parallel complexity up to a polynomial.)

The length of parallel computation is the sum over all lengths of sequential and parallel instructions executed during computation. The parallel complexity of the language will be the minimum over all complexities of parallel programs recognizing this language. We shall say, that a language L has parallel complexity $T(n)$ if there exists a parallel program accepting L, where all its accepting computations for a data of the length n have length not greater than $T(n)$.

Let PP-time denote the class of languages acceptable by a polynomial parallel program.

Finally assume, that relational system R includes effective arithmetic, that means arithmetic operations are polynomially programmable in R and all functions and predicates of R are polynomially programable on Turing machines. Then we can prove the following:

THEOREM 2

PP-time = P-space

where P-space is the class of languages acceptable by polynomial space bounded Turing machines.

Proof:

The proof consists of two simulations:

1. Every Turing machine, which uses polynomial space can be simulated by parallel program running in polynomial time.
2. Every polynomial time bounded parallel program can be simulated by a sequential one requiring polynomial amount of memory.

Part one

Let M be a q state one-tape deterministic Turing machine using $T(n)$ cells of memory for some polynomial T. Without loss of generality we can assume, that M has a two letter alphabet. Hence for a data of length n there exist at most

$$2^{T(n)} \cdot q \cdot T(n) \le 2^{p(n)}$$

different configurations of M, where $p(n)$ - is a polynomial. The polynomial time bounded algorithm simulating M can be written as follows:

```
*begin*
 *cobegin* c(i):= the i-th configuration of M□1≤i≤$2^{p(n)}$
        *coend;*
 *cobegin* *comment* use the next-move function of M;
        k(i):= the index of the next configuration
        after c(i)□1≤i≤$2^{p(i)}$ *coend;*
s := 1;
 *while* s≤p(n) *do*
   *begin* s := s+1;
     *cobegin* k(i) := k(k(i))□1≤i≤$2^{p(n)}$ *coend;*
   *end;*
 *if* k(1) is an index of accepting configuration
                    *then* accept *else* reject;
*end.*
```

To consider only the computations of the length $2^{p(n)}$ we define the successor of the terminal configuration as the same configuration. Indices k(i) computed after s steps of while loop code the configuration which follows after c(i) by application 2^s times next move function of M to c(i). The length of the computation M is bounded by $2^{p(n)}$, hence after p(n) steps of the loop we can simulate all possible computations of M in the memory bounded by T(n). Obviously the running time of our parallel program is of the range O(p(n)), that means it is bounded by some polynomial.

Part two

To complete the proof of the theorem it is sufficient to construct a polynomial space bounded nondeterministic Turing machine simulating an arbitrary given parallel program K running in polynomial time. Instead of writing a next-move function of the Turing machine, we will construct an algorithm easy by transformable to the formal one.

From the assumption about the relational system R we can rewrite our original program K into the parallel program K' with only boolean operations and 0-1 boolean variables by:

a) replacing every variable by 0-1 array coding the current value of this variable;

b) substituting for occurences of functors, predicates and arithmetic operations suitable polynomial programs;

c) substituting for every formula α occuring in the program K_α a new part of program K_α which carries out the value of α at a special boolean variable b_α.

Resulting parallel program K′ has exactly the same structure of parallelism as K and it remains polynomial (counting binary operations).

We will construct a recursive procedure find (i,t) which returns the value of i-th 0-1 variable after t steps of computation K′, assuming all variables and cells of arrays are numbered by integers. The steps are counted at external level, that means every parallel instruction is treated as one step and all binary instructions outside parallel instructions are counted separately. The idea of the procedure find is taken from [9]. By treating parallel instruction as a single statement our program is sequential. Build up the graph G whose vertices are conditions, substitutions and parallel instructions of K′. The edges are implied by the structure of K′. For the sake of simplicity assume that the terminal instruction is a successor of itself. A nondeterministic algorithm simulating K′ acts as follows:

1. Choose a path $p=p_1\ldots p_{T(n)}$ of the length T(n) in G such that p begins at the start vertex, where T(n) is a polynomial bounding time of K′. (It is done in T(n) nondeterministic steps.)

2. Compute find(0,T(n)) for the chosen path p. If the path p does not code the valid computation of K′ then procedure find will loop infinitely .

3. *if* find(o,T(n)) = 1 *then* accept *else* reject.

The proof will be completed if we construct polynomial space bounded algorithm for function find, since the acceptance of the input word by the program K is equivalent to find(0,T(n)) = 1, where 0 is the index of boolean variable denoting the acceptance of K′.

```
procedure find(i,t)

1. if t=0 then
      if i is an index of input variable then return $x_i$
                                        else return 0

2. if $p_t$ is not a parallel instruction at the choosen
   path  p  then
   a) if $p_t$ is a substitution which does not change $x_i$
      then return find(i,t-1);
   b) if $p_t$ is of the form $x_i := x_j \otimes x_k$ for some boolean
      operation $\otimes$ then return find(j,t-1) $\otimes$ find(k,t-1);
   c) if $p_t$ is a formula of the form $x_i = \alpha$ for $\alpha \in \{0,1\}$
      then compute find(i,t-1) and check if the edge in
      G  we have passed was in agreement with the struc-
      ture of  K'. If not then loop infinitely, other-
      wise return find(i,t-1);

3. if $p_t$ is a parallel instruction:
         cobegin $I_1 \Box \rho_1$ ,..., $I_r \Box \rho_r$  coend
   then
      Find  k  such that in $I_k$ $x_i$ can be changed.(There
      exists at most one such integer between 1 and r -
      it can be found having at the disposal procedure
      find(j,t-1) for pertinent j.)If such  k  does not
      exist then return find(i,t-1).
      Let $x_i$ be the variable occuring as a left side of
      substitution in $I_k$ at the instruction $p_t$. Then
      return find1(i,t,T(n))where procedure find1 is
      almost exactly the same as find, but constructed
      for "internal level" of program  K', that means
      for $I_k$ written in details, where each binary
      instruction is counted as a single instruction.
      Function find1 uses the second parameter  t  and
```

the function find when the third parameter decreases to 0. It plays the same role as input word for procedure find.

This completes the description of the algorithm. To observe that it operates in polynomial space note that in stack implementation of find the depth of recursion is bounded by $T^2(n)$ and the memory required for recording path p_t and all parameters at every level of recursion is also bounded by some polynomial.

This proves our theorem.

V. EXAMPLES OF PROGRAMS

Example 2

Boolean matrix multiplication in time O(logn). A is a n×n boolean matrix and n is a power of 2. Then after execution of program M matrix A contains its previous square.

```
M:
cobegin M(i,j,k):=A(i,k)∧A(k,j) □ 1≤i,j,k≤n coend;
s:=1;
while s≤log_2 n do
  begin s:=s+1;
    cobegin M(i,j,k):=M(i,j,2k-1)∧M(i,j,2k) □
            1≤i,j≤n∧1≤k≤n/2^(s-1)       coend
  end;
cobegin A(i,j):= M(i,j,1) □ 1≤i,j≤n    coend;
```

Example 3

G is the n vertex undirected graph with edges given by the n×n boolean matrix A. The following program L checks the connectivity of G in time $O(\log_2^2 n)$:

```
L:
cobegin A(i,i):=1 □ 1≤i≤n coend
for i:=1 step 1 until log₂n do A:= A × A;
comment A is the transitive closure of the incidence
        matrix for G
s:=1;
while s≤log₂n do
  begin cobegin A(i,j):=A(i,2j-1)∧A(i,2j) □
                1≤i≤n∧1≤j≤n/2^s          coend ;
  end;
s:=1; while s≤log₂n do
        begin cobegin A(i,1):=A(2i-1,1)∧A(2i,1) □
                      1≤i≤n/2^s      coend; s:=s+1 end;
if A(1,1) = 1 then accept else reject;
```

VI. FINAL REMARKS

In the section III we stated only rather evident logical properties of synchronous parallel programs. It seems that this kind of problems should be studied more carefully, particularly as a construction of the axiomatization and the system of inference rules for formulas of the form K_α, where K is a parallel program.

The proof of the complexity theorem based among others on the fact, that the value of an arbitrary variable after i steps of computation depends only at most on c^i other variables. There could be more active processors, but the history of computation for every variable is restricted to the exponential amount of memory. An interesting question arises, what would happen if functions had an unlimited number of arguments and were computable in one step.

The parallel language PP can be a useful tool for programming so far intractable problems, for instance members of P-space not known to be in P-time. The parallel algorithms for these problems will run in polynomial time.

REFERENCES

[1] Banachowski, L. et al: An introduction to algorithmic logic, in Mazurkiewich, A., Pawlak, Z. /Eds/, *Mathematical Foundations of Computer Science*, PWN, Warszawa, 1977.

[2] Chandra, A., Stockmeyer, L.: Alternation, *Proceedings of the 17-th Annual Symposium on Foundations of Computer Science*, Oct. 1976, pp. 98-108.

[3] Dańko, W.: Programs with arrays, *Fundamenta Informaticae*, N.3, 1978, pp. 379-398.

[4] Flynn, M.: Very High-Speed Computing Systems, *Proc. IEEE*, Vol. 54, Dec. 1976, pp. 1901-1909.

[5] Goldschlager, L.: Synchronous Parallel Computations, Technical Report No. 114, University of Toronto, 1977.

[6] Hartmanis, J., Simon, J.: On the Power of Multiplication in Random Access Machines, *Proceedings of the 15 th Annual Symposium on Switching and Automata Theory*, 1974, pp. 113-123.

[7] Hartmanis, J., Simon, J.: Structure of Feasible Computations, *Advances in Computer Science*, Vol. 14, Academic Press, N.Y., 1975, pp. 1-43.

[8] Kung, R.: The new method for finding solution of equations, in Traub, Wozniakowski (Eds), *Complexity and Effectiveness*, Prentice Hall, 1976.

[9] Pratt, V., Stockmeyer, L.: A Characterization of the Power of Vector Machines, *Journal of Computer and System Science*, Vol. 12, (1976), No 2,

[10] Radziszowski, S.: Programmability and P=NP conjecture, Karpinski, M. (ed), *Foundamentals of Computation Theory*, Lecture Notes in Computer Science, Vol. 56, Springer-Verlag, Berlin, 1977, pp. 494-498.

[11] Stapp, L.: The Proof of Correctness of Jacobi Parallel Program, CC PAS Report, No. 323, 1978.

S. Radziszowski
Uniwersytet Warszawski,
Instytut Informatyki
00-950 Warszawa PKiN,
POLAND