INFORMATICA, 1996, Vol. 7, No. 3, 311–336

COMPUTER ANALYSIS OF THE OBJECTIVE FUNCTION ALGORITHM

Gintautas DZEMYDA

Institute of Mathematics and Informatics
Akademijos 4, 2600 Vilnius, Lithuania
E-mail: dzemyda@ktl.mii.lt

Abstract. We consider a possibility of automating the analysis of a computer program realizing the objective function of an extremal problem, and of distributing the calculation of the function value into parallel processes on the basis of results of the analysis. The first problem is to recognize the constituent parts of the function. The next one is to determine their computing times. The third problem is to distribute the calculation of these parts among independent processes. A special language similar to PASCAL has been used to describe the objective function. A new scheduling algorithm, seeking to minimize the maximal finishing time of processing units, was proposed and investigated. Experiments are performed using a computer network.

Key words: Optimization,computer program analysis,parallel computing,scheduling.

1. Introduction. Extremal problems that arise in the design of technical systems can often be transformed into the form

$$\min_{X\in[A,B]} f(X),$$

where the objective function $f(X)$: $R^n \to R$ is continuous and multiextremal in the general case, $A = (a_1,\dots,a_n)$, $B = (b_1,\dots,b_n)$, $X = (x_1,\dots,x_n)$, $[A,B] = \{X\colon\ a_i \leqslant x_i \leqslant b_i,\ i = \overline{1,n}\}$.

The calculation of f value in optimization problems, occurring in scientific and engineering applications, often requires much expenditure of computing time. Sometimes the expenditure is so great that it is impossible to solve the problem by classical methods. In such cases it is reasonable to base optimization methods not only on the functional characteristic of the objective function (linearity, convexity, etc.) but also on the structure of a calculation process of the function value.

2. The main idea. Dzemyda and Tiešis [1] suggested to take into account the algorithmic structure of the objective function in local and global optimization algorithms which are oriented to a single-processor execution. These algorithms economize the computing time due to the coordinated calculation of function values at the nodes of a rectangular lattice [1] by storing and using the quantities that are common to several nodes.

The authors in [1] made use of

a) the known structure of function f:

$$f(X) = F(f_1(y_1), \dots, f_m(y_m)), \tag{1}$$

where $y_i \subseteq \{x_1, \dots, x_n\}$,

b) the location of trial points at the nodes of a rectangular lattice consisting of $\overline{L} = \prod\limits_{i=1}^{n} L_i$ nodes:

$$\begin{gathered}(\overline{x}_1 + s_1^{j_1}, \dots, \overline{x}_n + s_n^{j_n}),\\ s_i^{j_i} \in S_i \subset R,\ \exists s_i^{j_i} = 0,\\ j_i = \overline{1, L_i},\ i = \overline{1, n},\end{gathered}$$

where L_i is the number of discrete levels of the i-th coordinate of the lattice, $\overline{X} = (\overline{x}_1, \dots, \overline{x}_n)$ is the source point (node) of the lattice, S_i is a discrete set of L_i elements.

For example, if we need to calculate the values of function $f(x_1, x_2) = f_1(x_1) + f_2(x_2) + f_3(x_1, x_2)$ at four different points (x_1^*, x_2^*), (x_1^*, x_2^{**}), (x_1^{**}, x_2^*) and (x_1^{**}, x_2^{**}), it suffices to calculate four times the function f_3 only: functions f_1 and f_2 may be calculated only twice. If the numbers of variables and points are larger and the objective function is more composite, then the computational economy grows significantly.

The approach [1] showed good results because optimization algorithms often require to calculate values of the objective function at series of argument points which, e.g., should cover uniformly the definition area in global search or are specially located in the definition area for the evaluation of the gradient of function f in local optimization.

In such cases it is often reasonable to use parallel computing which finds wide applications in optimization. A particular type of parallelism in unconstrained optimization – simultaneous evaluation of the function to be minimized – is reviewed, e.g., in [20].

If we wish to calculate a fixed number of values of function f on a multiple processor computer, we may concurrently evaluate as many its values as the number of processors in the computer permits. But sometimes it isn't optimal because there might be better to calculate concurrently constituent parts of function f. Such situations arise when

• the calculation of the value of function f is computation-intensive,

• values of the function have to be calculated at the nodes of the rectangular lattice.

In the first case, parallel calculations allow us to optimize the load of processing units, and in the second one, the economy should be achieved by computing only once the quantities that are common to several nodes of the rectangular lattice.

In this paper, we consider a possibility of automating *the analysis* of a computer program that realizes the function f in order to recognize functions $f_1, f_2, \ldots, f_m$, and F, *the evaluation* of the computing time of constituent parts of the function f, and *the distribution* of calculations of a single value of the function f or of the set of values into p parallel processes. Different computers or a computer with several processors may be used for parallel calculations. A special optimization algorithm like that proposed in [1] or any other usual algorithm may be used for seeking the minimum of function f when its values are calculated in a parallel manner.

Our approach includes the following steps in solving the extremal problem:

1. Writing a program, that realizes the objective function, in a special PASCAL-based language.
2. Analysis of this program in order to recognize the constituent parts $f_1, f_2, \ldots, f_m$, F of the objective function.
3. Evaluation of the computing time of functions $f_1, f_2, \ldots, f_m$, F.
4. Making a schedule that distributes the tasks for calculating the values of functions $f_1, f_2, \ldots, f_m$ at the desired points of argument into p independent processes seeking to minimize the maximal finishing time of p processing units.
5. Parallel computing of the values of functions $f_1, f_2, \ldots, f_m$ at the desired points of argument.
6. Calculation of the value (or a set of values) of function F on the basis of values of the constituent parts $f_1, f_2, \ldots, f_m$ calculated in parallel.

Steps 1 – 3 should be executed before the optimization algorithm starts. The sequence of execution of steps 4 – 6 ought to be managed by the special optimization algorithm. Step 4 may be executed before the optimization algorithm starts if any stage of the optimization algorithm requires to calculate the values of the objective function at the nodes of rectangular lattices having identical configuration, i.e., having the same numbers $L_i,\ i = \overline{1,n}$.

We assume here that the computing time of any function $f_1, f_2, \ldots, f_m$ does not depend on the chosen point of argument.

Let us denote computing times of all the functions, composing the function f, by $t_1, \ldots, t_m$ and t_F. The calculation having the known values of functions $f_1, f_2, \ldots, f_m$ are used for computing F. Therefore, the computing time of function f will be equal approximately to $t_f = t_1 + \ldots + t_m + t_F$. The necessary conditions for our approach to be more effective are such:

- $t_1 + \ldots + t_m$ is considerably greater than t_F,
- $m \geqslant p$.

3. Analysis of computer codes. A special PASCAL-based language for description of the algorithm for calculating the function f at the point $X = (x_1, \ldots, x_n)$ has been developed. It is necessary to prepare a text of function f in this language. A special computer program analyzes this text, recognizes constituent parts $f_1, f_2, \ldots, f_m,\ F$ of function f (see (1)), converts this text into the PASCAL-program, and evaluates $t_1, \ldots, t_m$, and t_F.

The idea of analysing computer codes of programs, realizing scientific and engineering applications to their further solution in a parallel way, is not new. The reason is that such applications are very complex and computation-intensive. Since the scientific/engineering applications domain has been designated as the primary beneficiary of parallel processing, there have been several attempts to make parallel computers targeted for scientific/engineering applications [2–6]. The problem arises to estimate the extent of possible solution of the scientific/engineering application in a parallel way. Some early measurements of parallelism in FORTRAN programs are reported in [7]. These measurements were obtained by analysing the programs (statically) and determining which statements can execute in parallel because the most common method for exposing parallelism is to write the program in a conventional language (e.g., FORTRAN) and then detect the opportunities for parallel execution by a compiler. Moreover, the greatest advantage of using FORTRAN is that existing programs only have to be recompiled for new high-performance architectures (see [8]). The authors in [9] measured the total parallelism present in a FORTRAN program. This total parallelism can be observed if the program is executed on a computer which has unlimited processors and memory, does not incur any overhead in scheduling tasks and managing computer resources, does not incur any communication and synchronization overheads, and detects and exploits all the parallelism present in the program. Although an ideal computer which can exploit the total parallelism is not realizable, such measures are helpful in search for the optimal way of solving the problem.

An example of description of the algorithm for calculating the function f:

```
uses functions;
function f1; begin f1 := integration (x1, x2); end;
function f2; begin f2 := integration (x3, x5); end;
function f3; begin f3 := integration (x1, x7); end;
function f4; begin
  f4 := 9.60211* cos(x1)+ sqr (x2−0.1292*x1*x1+1.59155*x1−6)+10;
     end;
function f5; var s: real; begin
  differ (x2, x4, x7, s);
  f5 := 2*3.1414*s; end;
function f6; var i: integer; s: real;
  begin s := 0;
  for i := 1 to 10 do s := s + differ1(i, x5, x6);
  f6 := sqr(s); end;
function F; begin
  F := ((f1 + f2)/2 + f3* f4* f5))/f6; end;
```

The functions and procedures *integration, differ* and *differ1* are compiled in the TURBO-PASCAL unit *functions* seeking a shorter input program text for analysis. $x1, \ldots, x7$ are variables, $f1, \ldots, f6$ and F are the functions composing f (see (1)).

As a result of analysis of thc text of function f, some table is completed. An example of the table for the text above is shown in Table 1.

Table 1. The results of analysis of computer codes

Name of function	Name of variables	Names of variables	Computing time
$f1$	2	$x1$ $x2$	10
$f2$	2	$x3$ $x5$	17
$f3$	2	$x1$ $x7$	15
$f4$	2	$x1$ $x2$	24
$f5$	3	$x2$ $x4$ $x7$	7
$f6$	2	$x5$ $x6$	17
F	6	$f1$ $f2$ $f3$ $f4$ $f5$ $f6$	1

4. Distribution of calculations. The scheduling problem is to partition the tasks of calculating the values of functions $f_1, f_2, \ldots, f_m$ at a single point or at the set of argument points into p non-intersecting groups $A_1, \ldots, A_p$, i.e., to find a schedule how to calculate the values of functions composing the function f on p processing units.

Let us suppose we are given p (abstract) identical processing units P_i, $i = \overline{1,p}$, and a set of independent tasks $T = \{T_1, \ldots, T_r\}$ which is to be processed by the processing units. Let μ_j be the units of time required for completing T_j. Once a processor P_i begins to execute a task T_j, it works without interruption until the completion of that task, requiring altogether μ_j units of time. In a general case, it is required that the partial order $\prec$ on T be respected in the following sense: if $T_i \prec T_j$ then T_j cannot be started until T_i has been completed. But in our case $\prec$ is empty. The scheduling problem is as follows: find a schedule minimizing the maximum finishing time. However, various computer architectures and ways of exploiting parallelism influence the scheduling strategy, too (e.g., see [10, 14–17]). Therefore, the scheduling may be performed in several ways. The issues of concern and arising problems are summarized in [10]. NP-completeness of various restricted cases of the general scheduling problem is shown in [18, 19].

4.1. Scheduling. There are various scheduling criteria and strategies (e.g., see [11, 12]). The shortest-processing-time scheduling (SPT) [11] tends to reduce the mean number of unfinished tasks at each point in the schedule and to minimize the mean finishing time for p processing units. The largest-processing-time scheduling (LPT) [12] produces schedules which tend to maximize mean finishing time at each point in the schedule but minimize the maximum finishing time

$$C_A = \max_{L=\overline{1,p}} \sum_{T_l \in A_L} \mu_l. \tag{2}$$

Let

- SPT^* denote an SPT schedule which minimizes the maximum finishing time among all SPT schedules;
- OPT denote a schedule which gives the minimal possible value of C_A among all possible schedules;
- $t(S)$ be the maximum finishing time of schedule S.

In [12] the following best possible bounds are given:

$$1 \leqslant \frac{t(SPT^*)}{t(OPT)} \leqslant 2 - \frac{1}{p}, \qquad 1 \leqslant \frac{t(LPT)}{t(OPT)} \leqslant \frac{4}{3} - \frac{1}{3p}.$$

This means that in most cases LPT will be better than SPT^* in the sense of the maximum finishing time.

The LPT strategy minimizing the maximum finishing time is proposed and investigated by Graham [12]: a free processor always starts to execute the longest remaining unexecuted task. In our case, the scheduling algorithm is as follows (let us denote it by G):

- initially all groups $A_1, \dots, A_p$ are empty;
- find an unpartitioned task T_i whose execution time μ_i is the longest among unpartitioned tasks, and find a group A_k whose tasks have the shortest total execution time: $k = \arg \min_{L=\overline{1,p}} \sum_{T_j \in A_L} \mu_j$;
- put task T_i into the group A_k;
- the algorithm stops when there are no unpartitioned tasks.

In case of an arbitrary allocation of tasks into groups, Graham [12] noted two acceptable ways of minimizing the maximum finishing time:

- interchanging single tasks between two groups,
- moving a single task from its group to another one.

We propose below another criterion characterizing the scheduling quality and assisting in search for a schedule minimizing the maximum finishing time.

Let us seek a schedule such that the sums of execution times of the tasks composing each group be similar, i.e., the best partition (actually impossible) is

$$\sum_{T_l \in A_1} \mu_l = \sum_{T_l \in A_2} \mu_l = \cdots = \sum_{T_l \in A_p} \mu_l .$$

Let us analyse three equivalent criteria:

$$C_1 = \sum_{L=1}^{p} \sum_{k=L+1}^{p} \Big(\sum_{T_l \in A_L} \mu_l - \sum_{T_l \in A_k} \mu_l \Big)^2 ,$$

$$C_2 = \sum_{L=1}^{p} \Big(\sum_{T_l \in A_L} \mu_l - \overline{\mu} \Big)^2 , \text{ where } \overline{\mu} = \frac{1}{p} \sum_{j=1}^{r} \mu_j ,$$

$$C = \sum_{L=1}^{p} \Big(\sum_{T_l \in A_L} \mu_l \Big)^2 . \tag{3}$$

It is necessary to minimize these criteria seeking the best partition of $T_1, \ldots, T_r$.

REMARK 1. Partitions corresponding to the global minima of criteria C, C_1, and C_2 are the same.

The result of Remark 1 becomes obvious if we make the following transformations of C_1 and C_2:

$$C_1 = pC - p^2 \overline{\mu},$$
$$C_2 = C - (2p - 1)\overline{\mu}^2,$$

in which only C depends on the partition of tasks.

The criteria C, C_1, and C_2 extend a set of possible scheduling strategies, and allow to create special scheduling algorithms. C is simpler as compared to C_1 or C_2. Therefore, we shall describe the minimization of C more in detail.

Proposition 1. *Let the partition of tasks $T_1, \ldots, T_r$ into groups $A_1, \ldots, A_p$ be given. Let us analyse some task T_s. Let $T_s \in A_k$. The value of C will decrease after transferring T_s from the group A_k to the group A_L $(L \neq k)$ if*

$$\sum_{\substack{T_l \in A_k \\ (T_s \in A_k)}} \mu_l - \mu_s > \sum_{\substack{T_l \in A_L \\ (T_s \notin A_L)}} \mu_l . \tag{4}$$

Proof. Let a part of (3) containing the groups A_k and A_L be C^* before transferring T_s from the group A_k to the group A_L, and C^{**} after transferring T_s.

$$\begin{aligned}
C^* - C^{**} =& \Big(\sum_{\substack{T_l \in A_L \\ (T_s \in A_L)}} \mu_l \Big)^2 + \Big(\sum_{\substack{T_l \in A_L \\ (T_s \notin A_k)}} \mu_l \Big)^2 \\
& - \Big(\sum_{\substack{T_l \in A_L \\ (T_s \in A_L)}} \mu_l - \mu_s \Big)^2 - \Big(\sum_{\substack{T_l \in A_L \\ (T_s \notin A_L)}} \mu_l + \mu_s \Big)^2 \\
=& 2\mu_s \sum_{\substack{T_l \in A_k \\ (T_s \in A_k)}} \mu_l - 2\mu_s \sum_{\substack{T_l \in A_L \\ (T_s \notin A_L)}} \mu_l - 2\mu_s^2 \\
=& 2\mu_s \Big(\sum_{\substack{T_l \in A_k \\ (T_s \in A_k)}} \mu_l - \sum_{\substack{T_l \in A_L \\ (T_s \notin A_L)}} \mu_l - \mu_s \Big). \qquad (5)
\end{aligned}$$

Considering $\mu_s > 0$ in (5), it follows that

$$C^* - C^{**} > 0$$

when (4) is satisfied. The proposition is proved.

Proposition 2. *Let the partition of tasks $T_1, \dots, T_r$ into groups $A_1, \dots, A_p$ be given. Let us analyse two subgroups A_k^* and A_L^* of tasks from A_k and A_L ($L \neq k$, $A_k^* \subset A_k$, $A_L^* \subset A_L$). The value of C will decrease after interchanging A_k^* and A_L^* between the groups A_k and A_L if*

$$\sum_{\substack{T_l \in A_k \\ (A_k^* \subset A_k)}} \mu_l - \sum_{\substack{T_l \in A_L \\ (A_L^* \subset A_L)}} \mu_l > \sum_{T_l \in A_k^*} \mu_l - \sum_{T_l \in A_L^*} \mu_l > 0. \qquad (6)$$

Proof. Let a part of (3) containing the groups A_k and A_L be C^* before the interchanging of subgroups, and C^{**} after the interchanging.

$$\begin{aligned}
C^* - C^{**} =& \Big(\sum_{\substack{T_l \in A_k \\ (A_k^* \subset A_k)}} \mu_l \Big)^2 + \Big(\sum_{\substack{T_l \in A_L \\ (A_L^* \subset A_L)}} \mu_l \Big)^2 \\
& - \Big(\sum_{\substack{T_l \in A_k \\ (A_k^* \subset A_k)}} \mu_l - \sum_{T_l \in A_k^*} \mu_l + \sum_{T_l \in A_L^*} \mu_l \Big)^2 -
\end{aligned}$$

$$\begin{aligned}
&-\Big(\sum_{\substack{T_l\in A_L\\(A_L^*\subset A_L)}}\mu_l+\sum_{T_l\in A_k^*}\mu_l-\sum_{T_l\in A_L^*}\mu_l\Big)^2\\
&=2\sum_{\substack{T_l\in A_k\\(A_k^*\subset A_k)}}\mu_l\Big(\sum_{T_l\in A_k^*}\mu_l-\sum_{T_l\in A_L^*}\mu_l\Big)\\
&\quad-2\sum_{\substack{T_l\in A_L\\(A_L^*\subset A_L)}}\mu_l\Big(\sum_{T_l\in A_k^*}\mu_l-\sum_{T_l\in A_L^*}\mu_l\Big)\\
&\quad-2\Big(\sum_{T_l\in A_k^*}\mu_l-\sum_{T_l\in A_L^*}\mu_l\Big)^2\\
&=2\Big(\sum_{T_l\in A_k^*}\mu_l-\sum_{T_l\in A_L^*}\mu_l\Big)\Big(\sum_{\substack{T_l\in A_k\\(A_k^*\subset A_k)}}\mu_l\\
&\quad-\sum_{\substack{T_l\in A_L\\(A_L^*\subset A_L)}}\mu_l-\sum_{T_l\in A_k^*}\mu_l+\sum_{T_l\in A_L^*}\mu_l\Big).
\end{aligned}\tag{7}$$

If (6) is satisfied, then (7) implies that $C^* - C^{**} > 0$. The proposition is proved.

Proposition 3 follows as a partial case of Proposition 2.

Proposition 3. *Let there be given a partition of tasks $T_1, \ldots, T_r$ into groups $A_1, \ldots, A_p$. Let us analyse two tasks $T_s \in A_k$ and $T_j \in A_L$ $(L \neq k)$. The value of C will decrease after interchanging T_s and T_j between the groups A_k and A_L if*

$$\sum_{\substack{T_l\in A_k\\(T_s\in A_k)}}\mu_l-\sum_{\substack{T_l\in A_L\\(T_j\in A_L)}}\mu_l>\mu_s-\mu_j>0.$$

Minimization algorithms of C may be based on the following strategies:

- on the analysis of tasks in consecutive order and on the search for a group where to transfer a separate task with a view to decrease the value of C;
- on the analysis of the pairs of tasks from different groups for their further interchanging;
- on the analysis of the subgroups of tasks from different groups for their further interchanging.

The algorithms use different strategies of determining when the task T_s must be transferred from its group to another. For example, in the case of analysis of a separate task T_s (let $T_s \in A_k$), it may be transferred, e.g., into the group $A_L (L \neq k)$, where C decreases most, or into the first group found where (4) is satisfied. The realization of the third strategy is very computation-intensive. The algorithms stop when the transfer of any task, by the chosen strategy, does not decrease the value of C.

Two algorithms minimizing C are investigated.

Algorithm P1:

- initially all groups $A_1, \ldots, A_p$ are filled out using some algorithm of the initial partition of tasks;
- analyse tasks in consecutive order and search for a group A_L of transferring an individual task T_s (let $T_s \in A_k$, $L \neq k$) with a view to reduce the value of C: the transfer starts if

$$\sum_{\substack{T_l \in A_k \\ (T_s \in A_k)}} \mu_l - \mu_s > \min_{\substack{L=\overline{1,p} \\ L \neq k}} \sum_{T_l \in A_L} \mu_l;$$

- the algorithm stops when the transfer of any task, by the above strategy, does not decrease the value of C.

Algorithm P2:

- initially all groups $A_1, \ldots, A_p$ are filled out using some algorithm of the initial partition of tasks;
- analyse pairs of tasks $(T_s \in A_k, T_j \in A_L, L \neq k, s < j)$ in consecutive order and seek a pair of groups A_k and A_L for a possible interchange of the tasks T_s and T_j with a view to reduce the value of C: the interchange starts if

$$\sum_{\substack{T_l \in A_k \\ (T_s \in A_k)}} \mu_l - \sum_{\substack{T_l \in A_L \\ (T_j \in A_L)}} \mu_l > \mu_s - \mu_j > 0;$$

- the algorithm stops when the transfer of any task, by the above strategy, does not decrease the value of C.

Algorithms P1 and P2 require for the initial partition of tasks. Let us denote by U1 the following algorithm of initial partition:

- initially all groups $A_1, \ldots, A_p$ are empty;
- consider tasks $T_i, i = \overline{1,r}$, in consecutive order and put the current task T_c into the group having the smallest number of tasks.

Algorithm U1 is very simple and yields a schedule that is far from optimal.

REMARK 2. [12] yields the best possible bounds for U1 such as

$$1 \leqslant \frac{t(\mathrm{U1})}{t(OPT)} \leqslant 2 - \frac{1}{p}.$$

REMARK 3. Propositions 1 – 3 determine the necessary and sufficient conditions for the largest processing time to be reduced in the case of two groups of tasks.

REMARK 4. The algorithms, which minimize C by using the results of Propositions 1 – 3, also tend to minimize the maximum finishing time C_A (2), because at any step for arbitrarily selected pair $(A_i, A_j, i \neq j)$ of groups, they try to minimize the maximum finishing time $(\max\limits_{L=i,j} \sum\limits_{T_l \in A_L} \mu_l)$ of tasks in these two groups. The algorithms stop when it is impossible to reduce the maximum finishing time C_A by transferring the tasks between any pair of groups.

REMARK 5. Different schedules yielding different values of C may give the same value of C_A.

REMARK 6. If Graham's algorithm [12] is used as the initial partition of tasks for further analysis employing the algorithms based on Propositions 2 – 3, then the result of Graham's algorithm may be improved.

4.2. Results of the analysis of computing times. As a result of the analysis of computing times $\mu_1, \ldots, \mu_r$ of the functions $f_1, f_2, \ldots, f_m$ at various points of argument, a list of tasks for each processing unit P_i, $i = \overline{1,p}$, is completed. Note that $r \geqslant m$. In case of the above program, an example of the list of tasks for one of the processing units is shown in the first two columns of Table 2. The last column is filled out by the processing unit.

5. Experimental investigation.

The goal of experimental investigation was

- to estimate the efficiency of scheduling algorithms (Graham's (G), P1 and P2);
- to evaluate the case when it is reasonable to calculate the values of the objective function at the nodes of the rectangular lattice by using a computer network.

Table 2. The list of tasks

Name of function	Values of variables	Value of function
$f2$	0.5 0.5	1.5
$f2$	0.5 0.6	2.7
$f3$	0.7 0.7	5.4
$f3$	0.7 0.8	7.1
$f3$	0.7 0.9	5.2
$f5$	0.5 0.6 0.7	6.7
$f5$	0.5 0.7 0.7	9.5

5.1. Comparison of scheduling algorithms. Four strategies were investigated:

- algorithm G;
- algorithm U1+P1;
- algorithm G+P1;
- algorithm U1+(P1+P2).

The second and third strategies use algorithms U1 and G, respectively, for initial partitioning of tasks; further optimization is performed by P1. (P1+P2) means a combination of P1 and P2: P1+P2+P1+... as long as any transfer of a task from its group to another one or any interchange of two tasks from different groups do not reduce the value of C (and the value of C_A (see Remark 4)).

The experiments were carried out on the set of 100 randomly generated problems, and the results were averaged. $\mu_i \in [1, 100]$, $i = \overline{1, r}$, were generated at random, i.e., the situation, when the tasks having various execution times from the interval $[1, 100]$ appear with the same probability, was examined.

Remark 7. The experiments showed that

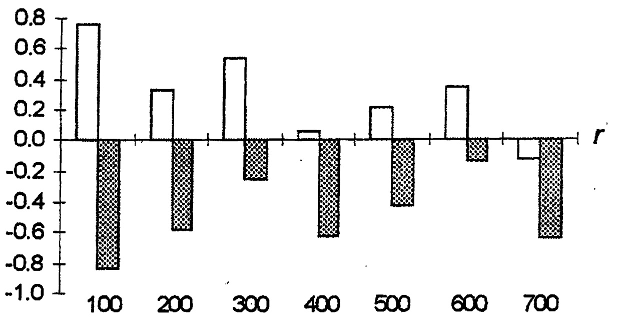
- algorithm G yields a partition of tasks which cannot be improved by P1;
- partitioning quality of algorithms G, U1+P1 and U1+(P1+P2) is similar; G yields a partition a bit better than that of U1+P1; U1+(P1+P2) yields a partition a bit better than that of G;
- algorithm U1+P1 operates significantly faster (about 15–200 times depending on the number of tasks) than U1+(P1+P2) because P2 is very

computation expensive;

- algorithm U1+P1 operates faster than G in the case of a small number p of processing units.

Figures 1 – 3 illustrate the conclusions of Remark 7.

a)



b)

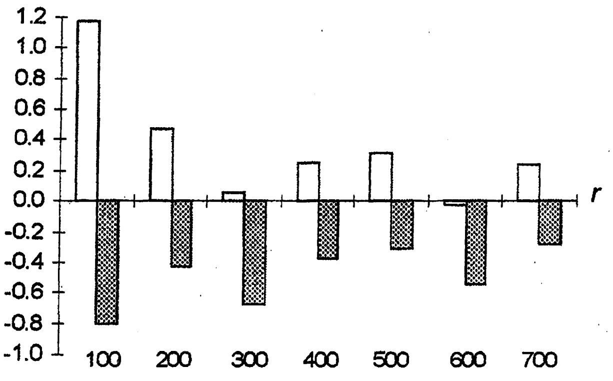


Fig. 1. The difference among values of C_A obtained by G, U1+P1 and U1+(P1+P2).

Fig. 1a illustrates the difference among the values of C_A obtained by G, U1+P1 and U1+(P1+P2), in case $p = 3$. Dark rectangles correspond to the difference C_A(U1+(P1+P2))-C_A(G), and white rectangles correspond to the difference C_A(U1+P1)-C_A(G), where C_A(S) denotes the value of C_A obtained by algorithm S. Taking into account that the obtained average values of C_A(U1+(P1+P2)), C_A(G), and C_A(U1+P1) during the experiments are equal to 6751.76, 6752.26, and 6752.56, respectively, we may conclude that the partitioning quality of these three algorithms is similar. Attempts to complicate tests by setting the values μ_i of five randomly selected tasks to be equal to 250 and generating other values of μ_i at random in $[1, 100]$ led to similar results as in Fig. 1a: the obtained average values of C_A(U1+(P1+P2)), C_A(G), and C_A(U1+P1) during the experiments are equal to 7082.19, 7082.68, and 7083.04, respectively (see differences in Fig. 1b).

The scope of algorithm G, in fact, is to search for a new order of tasks $T_i, i = \overline{1, r}$, where they are arranged in decreasing order of their execution time. Two simple sequential strategies for solving the problem of reordering the tasks are given in [13]. The authors in [13] also suggest, how to solve the above problem in a parallel manner using some processors. The first strategy from [13] is realized in G. This strategy is based on the interchange of pairs of tasks, and in [21, 22] it is called "bubble sorting". This strategy requires many paired comparisons of computing times. The way of optimizing computation expensiveness of G is to use a more sophisticated sorting. In our experiments, the algorithm of quick sorting by Hoare [23] was used, too.

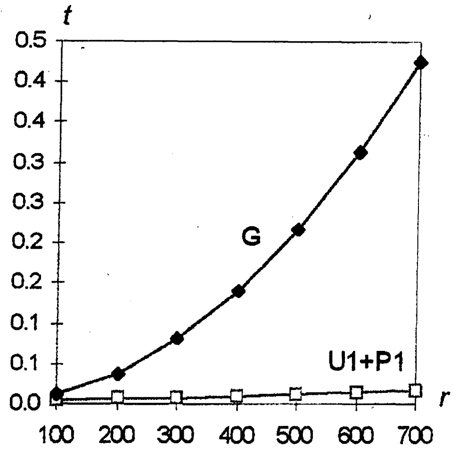


Fig. 2. Dependence of the average computing time t (in seconds) on the number r of tasks (G uses bubble sorting).

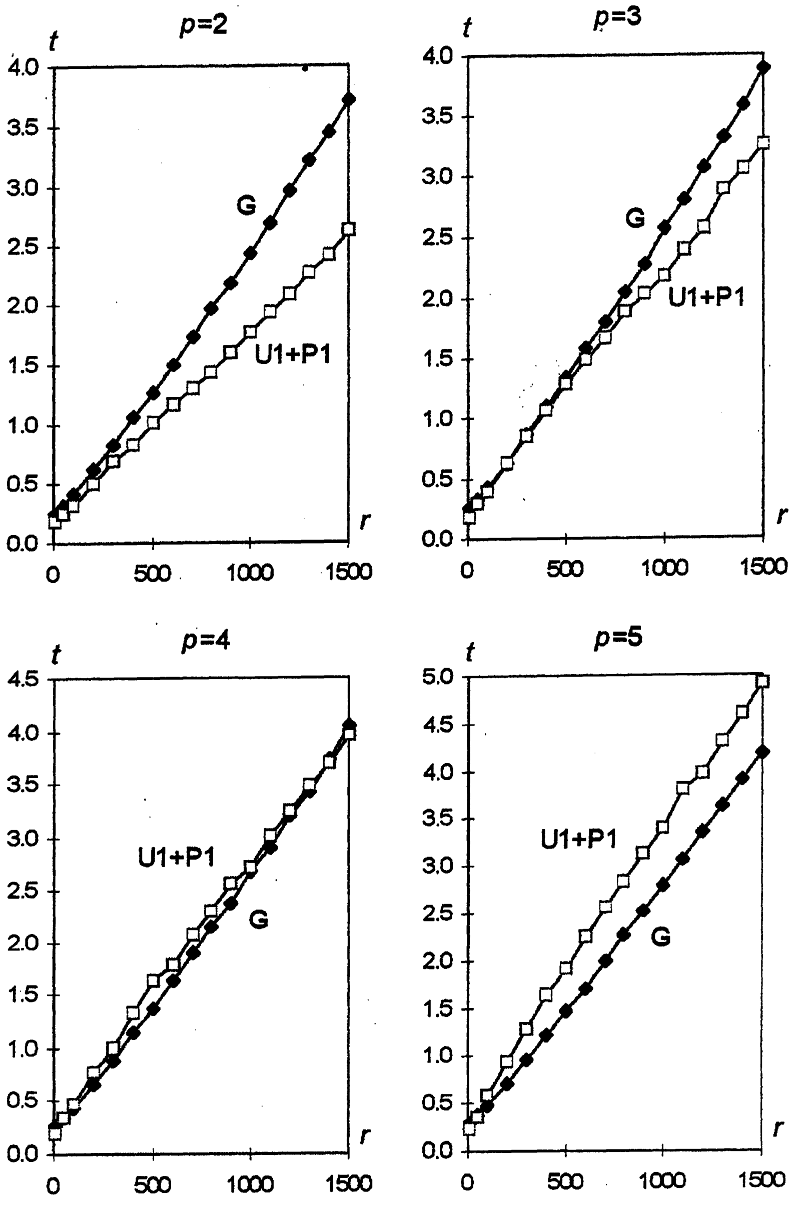


Fig. 3. Dependence of the average computing time t (in seconds) on the number r of tasks (G uses quick sorting) and on the number p of processing units.

Figures 2 and 3 show the dependencies of average computing time t (in seconds on 100 MHz PC) on the number r of tasks to be scheduled and on the number p of processing units. Fig. 2 illustrates the dependence of the average computing time t on r for G (using bubble sorting) and U1+P1, respectively. The value of p was set to be equal to 3 in this case. Fig. 3 illustrates the dependence of the average computing time t on r and p for G (using quick sorting) and U1+P1, respectively. Fig. 3 implies that the rapidity of algorithm U1+P1 depends on p essentially. The rapidity of G has a weak dependence on p because the main charge in G is put on the arranging of tasks in decreasing order of their execution time. Such a sorting is independent of p. The results indicate that the application of quick sorting in G is much faster than the bubble sorting, but algorithm U1+P1 is faster than G in both cases for small p (when $p = 2, 3$, and, in some cases, when $p = 4$ and even $p = 5$). Advantages of U1+P1 over G, which uses the quick sorting, are observed mostly for a larger number of tasks. This is essential, since the calculation of values of the objective function at the nodes of the rectangular lattice leads to a large number of tasks.

Algorithms P1 and P2 minimize both C and C_A. In Fig. 4 the dynamics of the best found values of $\sqrt{C}$ and C_A by P1 after the h-th transfer of task from one group to another is illustrated on the basis of the randomly generated problem ($\mu_i \in [1, 100], i = \overline{1,r}$, were generated at random, $r = 100$, $p = 3$). The minimization starts from the initial partition of tasks obtained by U1.

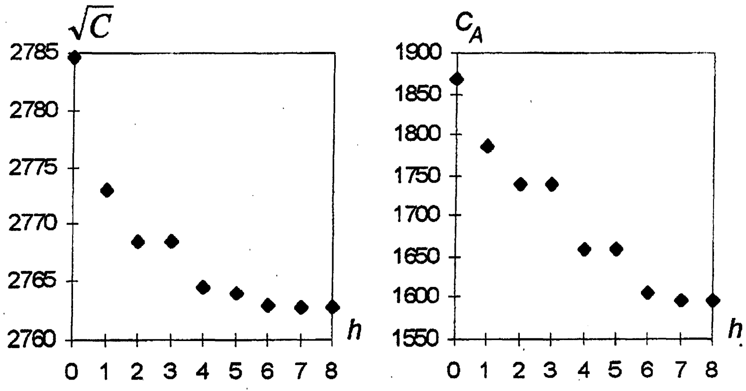


Fig. 4. Dynamics of the best found values of C and C_A by P1.

The experiments allowed to draw preliminary conclusions on the efficiency of algorithm U1+P1 as compared with G. Further theoretical investigations should provide more details. Solution of the scheduling problem is part of the search for minimum of the objective function (see Steps 1 – 6 in Section 2). In most cases, Steps 5 and 6 (calculation of values of the objective function) require much more computing time as compared to Step 4 (solving of the scheduling problem). So, even in the case of a small number p of processing units the usage of algorithm U1+P1 instead of G would not bring notable economy of computing time. Nevertheless, the results of this paper extend a set of criteria characterizing the scheduling quality for seeking a schedule minimizing the maximum finishing time, and a set of possible scheduling strategies.

5.2. Efficiency of parallel computing. Let us analyse a function of n variables involving functions that depend on all possible combinations of variables:

$$f^t(X) = \sum_{i_1=1}^{n} f_{i_1}(x_{i_1}) + \sum_{i_1=1}^{n} \sum_{i_2=i_1+1}^{n} f_{i_1 i_2}(x_{i_1}, x_{i_2})$$
$$+ \sum_{i_1=1}^{n} \sum_{i_2=i_1+1}^{n} \sum_{i_3=i_2+1}^{n} f_{i_1 i_2 i_3}(x_{i_1}, x_{i_2}, x_{i_3}) + \dots .$$

The goal is to calculate values of function f^t at $\overline{L}$ nodes of the rectangular lattice.

In the general case, the minimal number of functions depending on all possible combinations of variables $x_1, \dots, x_n$ may be defined as the following sum:

$$m = \sum_{i=1}^{n} \frac{n!}{i!(n-i)!}.$$

Let $L_i = L^*$, $i = \overline{1,n}$. If we need to calculate values of function f^t at $\overline{L} = L^{*n}$ nodes of the rectangular lattice, it is necessary to calculate

$$m^* = \sum_{i=1}^{n} \frac{L^{*i} n!}{i!(n-i)!}$$

values of functions composing the function f^t at different points of argument.

The values of function f^t may be calculated using four strategies:

S1) uncoordinated calculations using one processor;

S2) coordinated calculations using one processor;

S3) uncoordinated calculations using p processors;

S4) coordinated calculations using p processors.

Which of these four strategies is better depends on the computing times of functions composing the objective function.

Let

- τ_1 be the computing time of objective function values at all nodes of the rectangular lattice without taking into account that the trial points are the nodes of the lattice;
- τ_2 be the computing time of objective function values at all nodes of the rectangular lattice in the case of coordinated calculations of these values;

In [1] two criteria are introduced for comparison of the calculation strategies:

- *computing time economy:* $E = \tau_1/\tau_2$;
- *the number of objective function values calculated in time* τ_2 *in an uncoordinated way*: $N = \overline{L}\tau_2/\tau_1$.

Both criteria are related as follows: $N \cdot E = \overline{L}$.

Assume that

- computing times of any function composing the objective function are the same and equal to t_0;
- computing time of all the functions composing the objective function is considerably greater than summation of their computed values.

Thus, in a single-processor case

$$\tau_1 = t_0 \sum_{i=1}^{n} \frac{\overline{L}n!}{i!(n-i)!},$$

$$\tau_2 = t_0 \sum_{i=1}^{n} \frac{L^{*i}n!}{i!(n-i)!}.$$

In a p-processor case, τ_1 and τ_2 will be maximal finishing times of tasks scheduled for separate processors:

$$\tau_1 = t_0 \left[\text{int}\left(\sum_{i=1}^{n} \frac{\overline{L}n!}{i!(n-i)!}/p\right) + \text{mod}\left(\sum_{i=1}^{n} \frac{\overline{L}n!}{i!(n-i)!}, p\right)\right],$$

$$\tau_2 = t_0 \left[\text{int}\left(\sum_{i=i}^{n} \frac{L^{*i}n!}{i!(n-i)!}/p\right) + \text{mod}\left(\sum_{i=1}^{n} \frac{L^{*i}n!}{i!(n-i)!}, p\right)\right],$$

where the function $int(\nu)$ returns the integer part of argument ν, and the function $mod(v, p)$ returns 1 if the remainder part of division ν by p is nonzero.

Let us consider the following situation: $n = \overline{1,4}$; $L_i = L^* = \overline{1,5}$; $i = \overline{1,n}$; $p = \overline{1,3}$. The values of criteria N and E, in this case, have an insignificant dependence on the number of processing units p: the values of criteria vary in the range of some percent, and only in one case ($L^* = 2,\ p = 3,\ n = 2$) this percent is a bit greater than 10. Fig. 5 shows the dependence of criteria N and E on L^* and n in a single-processor case for the function f^t taking into account the above assumptions. Investigations of the single-processor computing econonly for a wide number of cases and assumptions on the structure of the objective function may be found in [1].

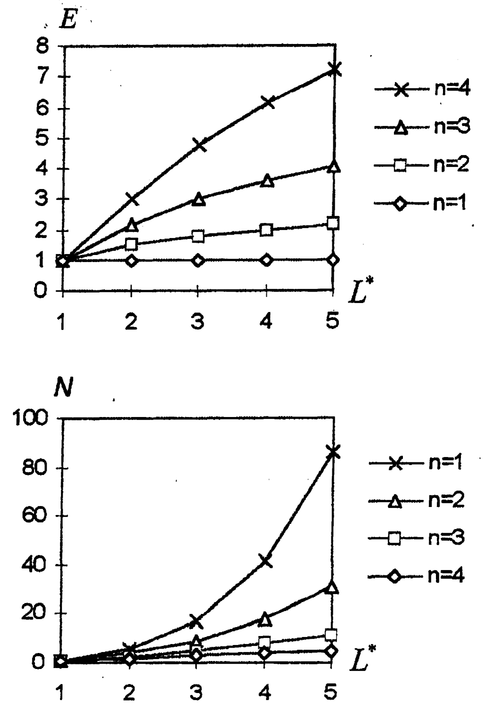


Fig. 5. Estimates of efficiency.

The relation of τ_1 and τ_2, in case $n = 4$, $p = \overline{1,3}$, $L_i = L^* = \overline{1,5}$, $i = \overline{1,n}$, is given in Fig. 6. Symbols '$\circ$' denote various values of L^*: from 1 on the left to 5 on the right.

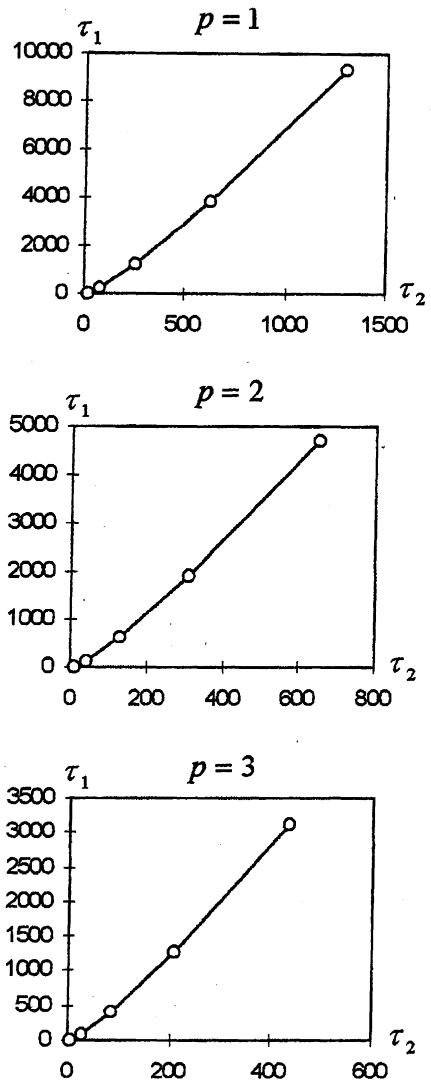


Fig. 6. Relation of τ_1 and τ_2.

The dependence of τ_2 on $L_i = L^* = \overline{1,5}$, $i = \overline{1,n}$, in case $n = 4$, $t_0 = 1$, $p = \overline{1,3}$, is illustrated in Fig. 7.

Computational experiments were carried out on a Windows for Workgroups based 66 MHz personal computer (PC) network. The scheme of the network

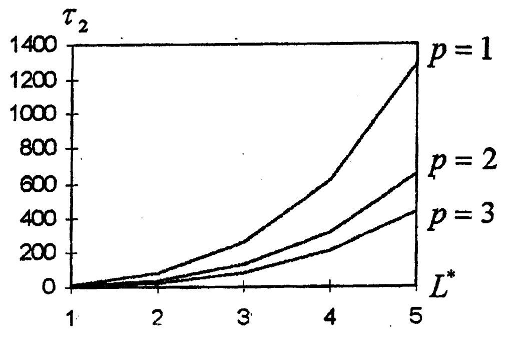


Fig. 7. Dependence of τ_2 on L^*.

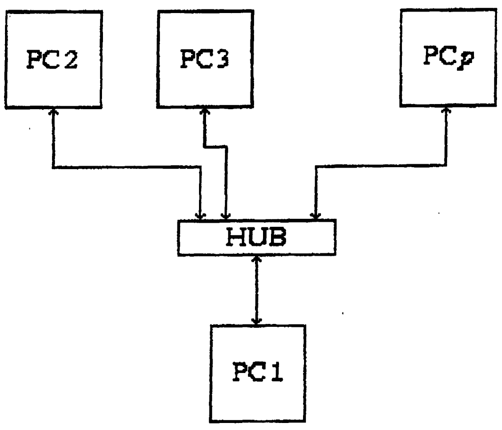


Fig. 8. Computer network.

is given in Fig. 8. Computers were connected using the hub "HP AdvanceS-tack 10Base-T Hub-8U". The goal of experiments was to estimate the computing expenditure on coordinated calculation of $\overline{L}$ values of function f^t in the case of fast calculation of functions composing the function f^t, i.e., when $t_0 = 0$. The rectangular lattice was defined as follows: $\overline{X} = (1, \ldots, 1)$, $S_i = \{0, 1, \ldots, L^* - 1\}$, $i = \overline{1, n}$. The dependence of computing time τ_2 (in seconds) on L^* and p is demonstrated in Fig. 9. Any point in Fig. 9 is obtained by averaging the results of 100 independent experiments. The results of Fig. 9 depend on the structure of data storing and the control of parallel calculations: most likely, they may be improved.

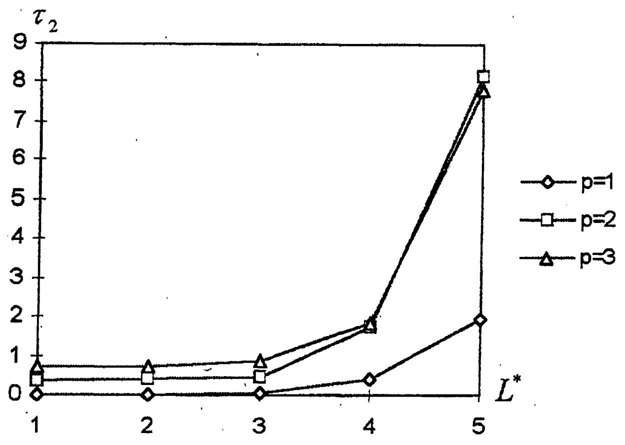


Fig. 9. Dependence of computing time τ_2 on L^* and p.

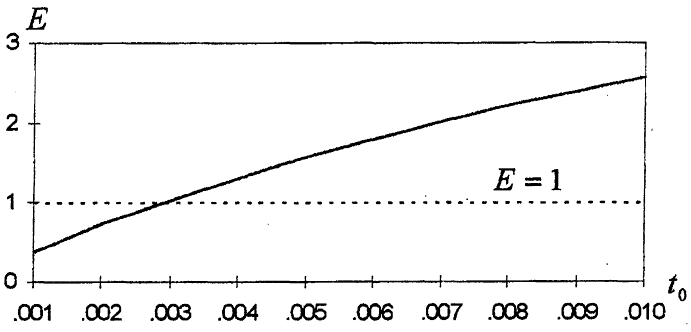


Fig. 10. The search for $E > 1$.

Figures 6, 7 and 9 allow us to estimate E and N in case $t_0 > 0$. Considering that, in our case, the management of independent calculations of $\overline{L}$ values of the function f^t requires much less time than that of the coordinated calculations, for any combination of L^* and p we may get the following estimates:

$$E \approx \tau_1^*/\tau_2^*,$$
$$N \approx \overline{L}\tau_2^*/\tau_1^*,$$

where

$$\tau_2^* = \tau_2(\text{from Fig. 9}) + t_0 \cdot \tau_2(\text{from Fig. 7}),$$
$$\tau_1^* = t_0 \cdot \tau_1(\text{from Fig. 6}).$$

For example, in case $L^* = 5$ and $p = 3$, the value of E becomes greater than 1 starting from $t_0 = 0.003$ seconds (see Fig. 10). It means that starting from $t_0 = 0.003$ the coordinated calculations become more effective than the uncoordinated ones.

6. Conclusions. The results of research proved the possibility of automating the analysis of the computer program realising the objective function of an extremal problem, the recognition of constituent parts of the function, and the distribution of calculations of its single value or the set of values into parallel processes.

Further investigations may be directed to

- the use of the proposed approach in other computing architectures;
- the extension of the fields of application.

REFERENCES

[1] Dzemyda, G., and V. Tiešis (1991). On the use of coordinated calculations in the solution of extremal problems. *Informatica,* **2**(2), 171–194.

[2] Beetem, J., M. Denneau and D. Weingarten (1985). The GF11 supercomputer. In *Proc. 12th Annual Symp. Comput. Architecture.* pp. 108–113.

[3] Crowther, W., *et al.* (1985). Performance measurements on a 128-node butterfly parallel processor. In *Proc. 1985 Conf. Parallel Processing.* pp. 531–540.

[4] Gurd, J.R., C.C. Kirkham and I. Watson (1985). The Manchester prototype dataflow computer. *Commun. ACM,* **28**, 35–52.

[5] Pfister, G.F., *et al.* (1985). The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. 1985 Conf. Parallel Processing.* pp. 764–771.

[6] Seitz, C.L. (1985). The cosmic cube. *Commun. ACM,* **28**, 22–33.

[7] Kuck, D.J. (1974). Measurements of parallelism in ordinary FORTRAN programs. *Computer,* **7**, 37–46.

[8] Kuck, D.J., R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe (1981). Dependence graphs and compiler optimizations. In *Proc. ACM Symp. Principles of Programming Languages.* pp. 207–218.

[9] Manoj Kumar (1988). Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers,* **37**(9), 1088–1098.

[10] Iannucci, R.A. (1988). Toward a dataflow/Von Neumann hybrid architecture. In *IEEE International Symposium on Computer Architecture*. pp. 131–140.

[11] Bruno, J., E.G. Coffman Jr. and R. Sethi (1974). Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, **17**(7), 382–387.

[12] Graham, R.L. (1969). Bounds on the multiprocessing timing anomalies. *SIAM J. Appl. Math.*, **17**(2), 416–429.

[13] Mani Chandy, K., and Jayadev Misra (1988). *Parallel Program Design: a Foundation.* Addison-Wesley Publishing Company.

[14] Lilja, D.J. (1991). *Architectural Alternatives for Exploiting Parallelism.* The Institute of Electrical and Electronics Engineers, IEEE Computer Society Press.

[15] Polychronopoulos, C.D., and D.J. Kuck (1987). Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, **C-36**(12), 1425–1439.

[16] Lam, M. (1988). Software pipelining: An effective scheduling techniques for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation.* The Association for Computing Machinery. pp. 318–328.

[17] Svenson (Ed.) (1985). *Introduction of Parallelism into Data Processing Algorithms.* Naukova Dumka, Kiev (in Russian).

[18] Ullman, J.D. (1975). NP complete scheduling problems. *Journal of Computer and System Sciences*, **10**, 384–393.

[19] Moiske, B. (1986). Scheduling problems on the evaluation of arithmetic expressions. In M. Feilmeler, G. Joubert and U. Schendel (Eds.), *Int. Conf. "Parallel Computing 85"*. Elsevier Science Publishers. pp. 383–387 .

[20] Lootsma, F.A. (1986). State-of-the-art in parallel unconstrained optimization. In M. Feilmeler, G. Joubert and U. Schendel (Eds.), *Int. Conf. "Parallel Computing 85"*. Elsevier Science Publishers. pp. 157–163.

[21] Knuth, D.E. (1973). *The Art of Computer Programming*, Vol. 3. Addison-Wesley.

[22] Wirth, N. (1976). *Algorithms + Data Structures = Programs.* Prentice-Hall.

[23] Hoare, C.A.R. (1971). Proof of recursive program: Quicksort. *Computer Journal*, **14**(4), 391–395.

Received October 1996

G. Dzemyda received his Ph.D. degree from the Kaunas Polytechnic Institute, Kaunas, Lithuania, in 1984. He is a senior researcher at the Optimization Department of the Institute of Mathematics and Informatics, and an Associate Professor at the Vilnius Pedagogical University. His research interests include interaction of optimization and data analysis.

TIKSLO FUNKCIJOS ALGORITMO KOMPIUTERINĖ ANALIZĖ

Gintautas DZEMYDA

Straipsnyje parodyta galimybė automatiškai analizuoti optimizavimo uždavinio tikslo funkcijos kompiuterinę programą ir paskirstyti tos funkcijos reikšmės skaičiavimą lygiagrečiai keliems kompiuteriams. Paskalio tipo kalba panaudota funkcijos aprašymui. Eksperimentams panaudotas kompiuterių tinklas.