ON CHOOSING A PLAN FOR THE EXECUTION OF DATA FLOW PROGRAM GRAPH

INFORMATICA 3/86

Borut Robič, Jurij Šilc
Jožef Stefan Institute, Ljubljana

UDK: 681.519.7

ABSTRACT - In the paper we present a generalized analisys of static data flow program graphs. These graphs are allowed to have nodes that use more than one unit of time for their execution. Such graphs are more realistic then graphs with nodes that execute in one unit of time. We restrict our consideration to graphs with integer execution times of their nodes. In the paper we first briefly describe the data flow concept of computation. Next we describe the basic data flow architecture and a common way of the execution of a graph on it. We show that this way of the execution has a drawback. In the next sections we introduce the notion of a static data flow program graph and describe the state of the execution of such graph. The state consists of a few time depending sets of nodes. We define a plan for the execution of a program graph which is the result of the analysis of the graph made before its execution. There are two special plans for the execution. Information, obtained by these two plans is used for defining the third special plan, which we call the heuristic plan for the graph execution. The execution of a graph according to this plan may minimize the number of processors needed, without lengthening the total execution time of the graph. Finally, we informally describe the algorithm for obtaining plans for the execution.

O IZBIRANJU NAČRTA ZA IZVRŠEVANJE PODATKOVNO PRETOKOVNIH GRAFOV - V delu je podana posplošena analiza statičnih podatkovno pretokovnih grafov. Točke takšnih programskih grafov se lahko izvršujejo poljubno celo število časovnih enot. Uvodoma je opisan koncept podatkovno pretokovnega računanja. Sledi opis značilne podatkovno pretokovne arhitekture ter enega izmed možnih izvrševanj podatkovno pretokovnega grafa na njej. Prikazana je slabost takšnega načina izvrševanja. Po opisu statičnega programskega grafa sledijo definicije množic točk, ki sestavljajo stanje izvrševanja grafa. Definiran je načrt izvrševanja programskega grafa, ki je rezultat njegove predhodne analize. V splošnem obstaja več načrtov za izvrševanje. Izvrštve, ki ustrezajo posameznim načrtom, se razlikujejo po uporabljenem številu procesnih elementov in ne podaljšujejo minimalnega časa, potrebnega za izvršitev programskega grafa, v kolikor je na voljo dovolj procesnih elementov. Obstajata dva posebna načrta za t.i. takojšnje in leno izvrševanje, ki v splošnem ne minimizirata števila potrebnih procesnih elementov. Ker sistematično pregledovanje vseh možnih načrtov vodi v kombinantorično eksplozijo, je v delu predlagan hevristični postopek za izbiro načrta izvrševanja, ki teži k minimizaciji števila procesnih elementov.

1. INTRODUCTION

A data flow system comprises a user-oriented high-level language, a low-level base language, and a highly-parallel computer architecture. User programs are written in the high-level language and are translated into corresponding programs in the base language. A base language program is a graph composed of nodes interconnected via directed arcs. Each internal node is an operation and represents a separate processing elemet capable of accepting, processing and emitting value tokens travelling along the graph arcs. Each operation executes only when all tokens, carrying operands required by that operation have arrived. At that point the operands are consumed by the node and new tokens, carrying results, are placed on the output arcs [COM82]. This fundamental principle permits the graph to be mapped onto a computer architecture consisting of a very large number of independent processing elements and switches, able to connect any two processing elements, making a data path between them. Separate data paths can cross the switch simultaneously [KFG584]. For example see Fig.1.

2. SIMULATION OF DATA FLOW ARCHITECTURE

In general not all operations (nodes of a graph) need to be assigned to processing elements at the same moment since not all operations have available all input operands at that moment. To make the data flow concept possible even without a very large number of processing elements data flow computers are

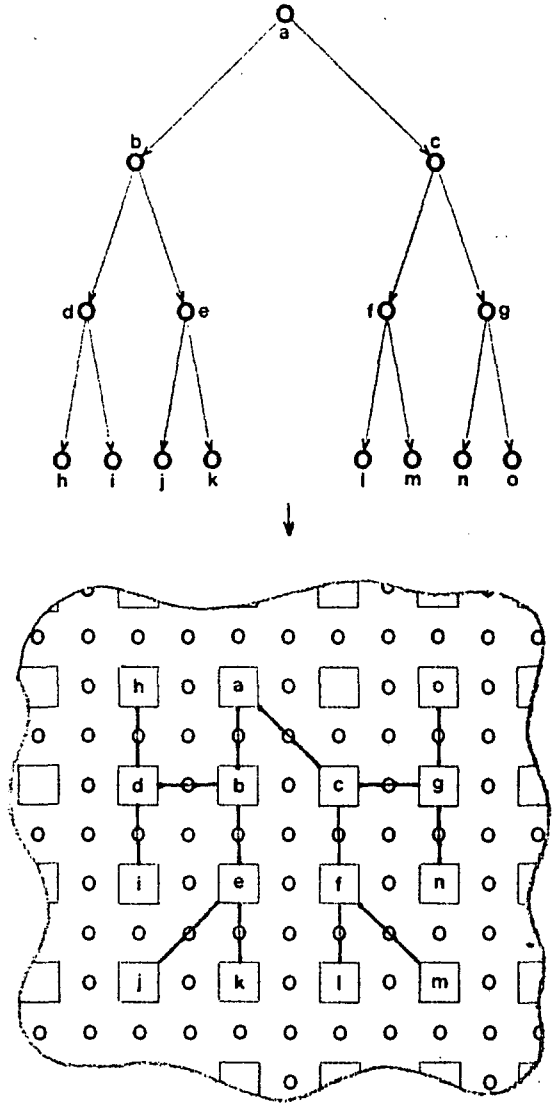


Fig.1: In the switch lattice the squares represent processing elements, circles represent switches, lines represent data paths.

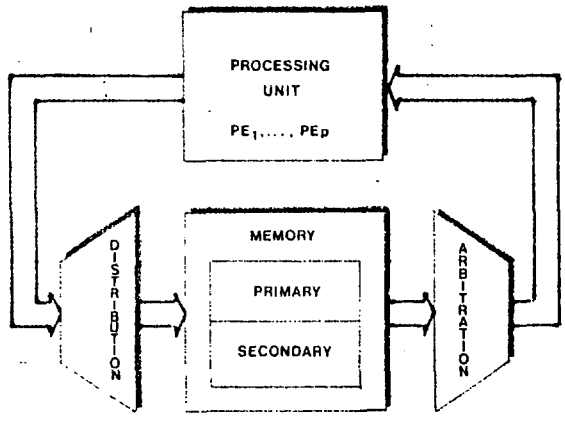


Fig.2: The model of data flow machine.

usually based on a packet communication machine organization, consisting of a circular instruction execution pipeline of resources. The resources are memory, arbitration, processing and distribution unit. The memory unit is divided into the primary and secondary part, the former beeing faster. The processing unit consists of a number of processing elements (Fig.2).

Nodes, having none of their input operands arrived reside in the secondary memory. These are noncreated nodes. At the moment when the first input operand has arrived the node is created, i.e. moved to the primary memory to wait for the other input operands. A created node becomes executable at the moment when the last input operand has arrived. Executable node is ready for the execution and may be transferred (fired) to the processing unit where a processing element starts to execute it. It is the arbitration unit that decides which of the nodes are executable and which processing elements are to be allocated to. The place in the memory unit which was occupied by that node is now set free. While the node is beeing executed the distribution unit finds out where the nodes waiting for the result are. When the result is produced, the distribution unit sends it to all nodes waiting for it, creating some of them if neccessary. The node that has been executed is now deleted.

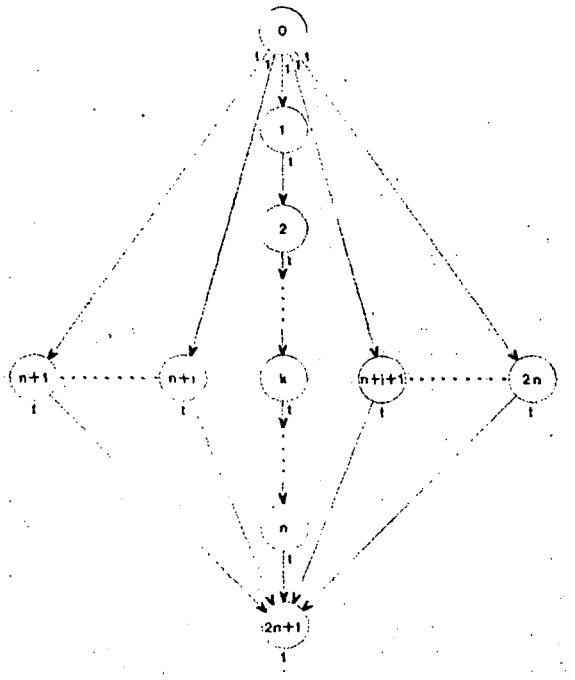
Such execution may need more processing elements than there are in the processing unit. The problem where there are more executable nodes than processing elements must be solved during the execution of a graph. All these nodes are executed in several steps implying the lengthening the total minimum execution time of a graph. Note that after each such step again the same problem may appear.

An anaysis of the graph before its execution may prevent the problem discussed above. The basic observation is that in general not all executable nodes have to be fired immediately, since some of them may wait a period of time in the memory without lengthening the total minimum execution time of a graph. Analysing the graph we obtain several plans for its execution, each execution having its own characteristic. Information obtainded by the plan is used by the architecture components during the actual execution of a graph in deciding which executable nodes should be retained. The execution according to a properly chosen plan may result in minimization of some resources needed, such as the number of processing elements. We point out three special plans for the execution. Execution according to the first of them is essentially the immediate execution, described above. The execution according to the second plan is the opposite of the immediate execution, while the execution according to the third plan often uses minimum number of processing elements. Executing graphs according to this plan we may avoid the problems discussed above.

3. STATIC DATA FLOW PROGRAM GRAPH

There are two ways of envisaging a data flow program: as a static or as a dynamic data flow program graph. We limit our discussion to static graphs. In short, static data flow program graphs are acyclic, while the dynamic are not.

We define a static data flow program graph G = (V,A), in further discussion program graph, to be a directed, acyclic, and simple (no multiple arcs of the same direction between two nodes) graph. The set V of nodes is partitioned into three disjoint sets V_S ,V_I , V_F of starting, internal and final nodes, respectively. The starting nodes have no input archs while the final nodes have no output archs. Furthermore, there must always exist a path to any internal or final node from some starting node and, similarly, a path from any starting or internal node to some final node. Starting nodes are used to carry the input values while the final nodes store the results. Internal nodes carry operations [Ro586]. The time of the execution of a node n is t_n, where t_n = 1 for all starting and final nodes n.



$\mu_{imm} = \mu_{lazy} = n+1 \qquad \mu_{heu} = 2$

Fig.3: Static data flow program graph.

The length of a path is defined to be the sum of the execution times of the nodes along the path. The path from a set of nodes A to a set of nodes B is defined to be any path that starts at some node of the set A and ends at some node of the set B. The longest path from the set A to the set B is a path that has among all the paths from the set A to the set B maximal length. The length of the longest path from the set A to the set B is described by l(A,B). When the set consists of only one element we substitute the set by its element. For example, $l(n,V_F)$ is the length of the longest path from the node n to the set of the final nodes. Note that since the program graph is acyclic by assumption, the lengths l(m,n) for any pair of nodes can easily be computed using one of well known algorithms [Law76]. The evaluation of l(A,B) is then trivial for any two disjoint sets of nodes A and B.

If a node n is beeing executed at the moment j we define $l'(n,V_F)$ to be the sum of the length of the longest path from the set of its sons to the set of the final nodes and, the time neccessary for the node n to finish its execution.

4. THE STATE OF THE GRAPH EXECUTION

The state of the execution of a graph at the moment j is the quintuple $\sigma_j = (N_j, C_j, E_j, F_j, D_j)$ where N_j ,C_j ,E_j ,F_j and D_j are the sets of noncreated, created, executable, executing and deleted nodes at the moment j, respectively. There are also few other sets used for computing the sets mentioned. The sets are described bellow:

- F_j^* ... (executed nodes) is the set of all the nodes fired before the moment j that have finished their execution at the moment j and put their results on all output arcs,
- C_j^{new} ... (new created nodes) is the set of all those nodes that received at the moment j the first input operand,
- C_j^{old} ... (old created nodes) is the set of all those nodes that had been created at any of the moments before the moment j yet did not fire until the moment j-1 including,
- C_j ... (created nodes) is the union of old and new created nodes,
- E_j^{new} ... (new executable nodes) is the set of all those nodes that have received the last input operand at the moment j,
- E_j^{old} ... (old executable nodes) is the set of all those nodes that had become executable before the moment j but have not fired until the moment j-1 including,
- E_j ... (executable nodes) is the union of the old and new executable nodes,
- E_j^* ... (unconditionally executable nodes) is the set of all those executable nodes that must be fired at the moment j so as not to lengthen the total execution time;
- E_j^v ... (conditionally executable nodes) is the set of all those executable nodes that are fired at the moment j although they could be fired later withot lengthening the total execution time of a graph,
- F_j^{new} ... (new executing nodes) is the set of all the nodes that started executing at the moment j,
- F_j^{old} ... (old executing nodes) is the set of all those nodes that had been fired before the moment j yet have not finished their excution till the moment j including,
- F_j ... (executing nodes) is the union of the old and new executing nodes,
- N_j ... (noncreated nodes) is the set of all those nodes that have not created till the moment j including,
- D_j ... (deleted nodes) is the set of all those nodes that executed till the moment j including,
- E_j^{cr} ... (critical nodes) is the set of all those nodes that allways become unconditionally executable at the same moment j regardless of the plan of the execution.

5. THE PLAN FOR THE EXECUTION

The plan of the execution is a supervisor that controls the execution of a program

graph. Consequently, the plan implyes a certain degree of control flow in data flow architecture and is realized by a time control vector associated to each node of a program graph.

The plan for the execution of a graph is determined by the rule which selects the sets E_j^v. There are two trivial plans for the execution of a graph. These are a plan for immediate and a plan for lazy execution. The plan for immediate execution is determined by choosing E_j^v to consists of all those executable nodes at the moment j that could be fired even later. The plan for the lazy execution is determined by forcing E_j^v to be empty at all moments j. Consequently, the immediate execution fires the nodes as soon as possible while the lazy execution defers firing to the last possible moment. In general, none of these two executions minimizes the number of processing elements needed. The plan for the execution that uses minimum number of processing elements could be found by sistematic examination of all possible rules for the choosing the sets E_j^v. Since we want to avoid the combinatorial explosion we try to find a heuristic rule such that the execution according to the associated heuristic plan would use the number of processing elements as close as possible to the theoretical lower bound. For example, execution of the graph on Fig.3 according to the immediate plan needs n+1 processing elements, since at the moment j = 1 the node 1 is fired simultaneously with the nodes n + i, 1≤i≤n. Similarly, the lazy plan implies the execution that involves n+1 processing elements, too. Namely, at the last step the node n is fired together with the nodes n + i, 1≤i≤n. On the other hand, the heuristic plan offers an execution with only two processing elements for at each step only the pair of nodes k, and n+k are beeing executed.

The plan for the heuristic execution will be discussed below.

6. THE TIME CONTROL VECTOR

Every node n is associated with a time control vector $\tau_n = (\tau_{C,n}, \tau_{E,n}, \tau_{F,n})$.

$\tau_{C,n}$, $\tau_{E,n}$ and $\tau_{F,n}$ are the moments when the node n becomes created, executable and fired. Time control vectors are computed for each node while the plan for the execution is beeing constructed. Since all the sets described above are affected by the rule for choosing the set E_j^v, the time control vectors depend on a plan constructed. Every plan for the execution has its own set of time control vectors. To point out that the time control vector of a node n is computed according to the plan for immediate or lazy execution we write τ_n^{imm} or τ_n^{lazy}, respectively.

7. THE PLAN FOR HEURISTIC EXECUTION

7.1. CRITICAL NODES

Let $\tau_n^{imm} = (\tau_{C,n}^{imm}, \tau_{E,n}^{imm}, \tau_{F,n}^{imm})$ be time control vectors of a node n according to immediate and $\tau_n^{lazy} = (\tau_{C,n}^{lazy}, \tau_{E,n}^{lazy}, \tau_{F,n}^{lazy})$ time control vectors of a node n according to lazy execution. Then we say that an internal node n is critical if $\tau_{F,n}^{imm} = \tau_{F,n}^{lazy} = \tau_{F,n}^{cr}$.

THEOREM 1.

Every critical node n is fired at the moment $\tau_{F,n}^{cr}$, regardless of the plan of the execution.

PROOF: $\tau_{F,n}^{imm}$ is the first possible moment when the node n can be fired, regardless of the plan of the execution of a program graph. On the other hand, $\tau_{F,n}^{lazy}$ is the last moment when the node can be fired without lengthening the total execution time, taken over all possible plans of the execution. Consequently, if n is a critical node, then $\tau_{F,n}^{imm} = \tau_{F,n}^{lazy} = \tau_{F,n}^{cr}$ which means, that the moment $\tau_{F,n}^{cr}$ is the only moment when the node n can be fired in all possible plans of the executions. Q.E.D.

LEMMA 1.

There is at least one critical node in the program graph.

PROOF: Let be p the longest path from the set of starting nodes to the set of final nodes. Then, in any plan of the execution the son of starting node on the path p must be fired at the moment j=1 so as not to lengther the total execution time. Consequently, the son of the starting node is critical. Q.E.D.

THEOREM 2.

A node is critical if and only if it is on a longest path from the set of starting nodes to the set of final nodes.

PROOF: Let p be some longest path from the set V_S to the set V_F. By Lemma 1 the son of the starting node on the path p is critical. Now suppose that all first $k > 0$ internal nodes $n_1, n_2, \ldots, n_k$ on the path p are critical. Consequently, the first moment at which the node n_{k+1} can be fired is the moment $1+\Sigma_{i=1}^{k} t_{n_i}$. Since the node n_{k+1} is on the longest path from the set V_S to the set V_F this is also the last moment, when it can be fired, so as not to lengthen the total execution time of the graph implying that the node n_{k+1} is critical. Conversely, suppose a node n is critical. Consider the longest path p of the all paths from V_S to V_F, passing the node n. We define the subpath p' of the path p to be the path consisting of all the nodes from V_S to the father of the node n. Similarly, the subpath p" of the path p consists of all the nodes from the son of the node n to the set V_F. We define $l(p')$ and $l(p")$ to be the lengths of the paths p' and p", respectively. Note, that since p is the longest path from V_S to V_F through the node n, the path p' must be the longest of the paths from V_S to the set of fathers of n. Similarly, the path p" must be the longest path from the set of sons of the node n to the V_F. Suppose the relation $l(p')+t_n+l(p") < l(V_S, V_F)$ holds. Then there must be some node m on the path p, that can be fired at at least two different moments. The node m must be either on the path p' or on the path p", since the node n is critical, by assumption. Were the node m on the path p' the node n could be fired at

at least two different moments contradicting the assumption that it is critical. If the node m were on the path p", a similar argument would result. Consequently, relation $l(p')+t_n+l(p'') = l(V_S, V_F)$ is true, implying that the path p is one of the longest paths from V_S to V_F. Q.E.D.

Let E_j^{cr} be the set of all critical internal nodes at the moment j. We call the value

$\zeta = \max |E_j^{cr}|$, where $1 \le j \le t^*$, the maximal critical parallelism of a program graph, where $t^* = l(V_S, V_F) - 1$.

Constructing the longest paths from V_S to V_F [Law76] and considering their nodes, the critical internal nodes are easily computed. We could determine the critical nodes also by computing the time control vectors associated to the plan for the immediate and lazy execution and comparing their third components, for each internal node.

7.2. UPPER AND LOWER BOUNDS ON μ_{min}

Let us define μ_{min} to be the minimum number of processing elements needed for the execution of a program graph, taken over all possible plans of the execution. We want to bound μ_{min} as much as possible.

Let μ_{imm} and μ_{lazy} denote the number of processing elements needed in the immediate and lazy execution of a program graph, respectively. We define the average parallelism of a program graph to be $\pi = t_{seq}/t^*$, where $t_{seq} = \sum_n t_n$, $n \in V_I$, and maximal parallelism to be $\lceil \pi \rceil$*. We also define $\alpha = \max(\lceil \pi \rceil, \zeta)$, and $\beta = \min(\mu_{imm}, \mu_{lazy})$. Then the following theorem illustrates the upper and lower bounds on μ_{min}.

THEOREM 3.

Minimal number of processing elements needed is bounded by $\alpha \le \mu_{min} \le \beta$.

PROOF: In case of $\zeta \ge \lceil \pi \rceil$ the proof is evident since there exists a moment j when at least ζ critical nodes must be fired. Suppose now that $\zeta < \lceil \pi \rceil$. Let μ_j, $j = 1,2,\ldots,t^*$ denote the number of internal executing nodes at the moment j. Then $\sum_{j=1}^{t^*} \mu_j = t_{seq}$. Suppose that $\mu_j < \pi$, for all $j=1,2,\ldots,t^*$. Then we have $t_{seq} = \sum_{j=1}^{t^*} \mu_j < \sum_{j=1}^{t^*} \pi = t_{seq}$, a contradiction. Consequently, $\mu_{j'} \ge \pi$, for some j', $1 \le j' \le t^*$. Since the number $\mu_{j'}$ is an integer, we also have $\mu_{j'} \ge \lceil \pi \rceil$, and a fortiori $\mu_{min} \ge \lceil \pi \rceil$. The right side of the nonequation is also evident. Q.E.D.

If we get $\alpha = \beta$ then at least one of the plans for immediate or lazy execution is also an optimal one. If $\alpha < \beta$, which is much more possible, we can try to push the upper bound β down by additional computations where we chose another plan of the program graph execution. For example, we may try to choose the sets E_j^v on each step in such a way, that $|F_j - V_F|$, $1 \le j \le t^*$, is as close as possible to α. The details are discussed in next section.

* $\lceil r \rceil$ is the lowest integer, greater or equal to r.

7.3. THE PLAN FOR HEURISTIC EXECUTION

Since the lower bound on the minimum number of processing units needed is given by α, we choose the sets E_j^v in such a way, that the number $|F_j - V_F|$ is on each step j, $1 \le j \le t^*$, as close as possible to α. To do this we first consider the number $c = |E_j^* - V_F| + |F_j^{old}|$ of the processors that are already needed at the moment j. If this number is greater than or equal to α, nothing is put into the set E_j^v. On the other hand, if $c < \alpha$, we consider the executable nodes, that need not to be fired at the moment j, i.e. the set $E_j - E_j^*$. If there are at most $m = \alpha - c$ such nodes we put all of them into the set E_j^v. However, if there are more than m such nodes n, we choose m of them, having the highest values $l(n, V_F)$, and put them into the set E_j^v (Fig.4).

```
c := |E*j - VF| + |F^old_j| ;
if c ≥ α then E^v_j := ∅
else
  begin
    m := α - c ;
    if |Ej - E*j| < m  then  E^v_j := Ej - E*j
    else Choose m nodes n from Ej - E*j having
         the longest values l(n,VF) and put
         them into the set E^v_j ;
  end;
```

Fig.4: Heuristic choosing the sets E_j^v.

8. ALGORITHM FOR OBTAINING EXECUTION PLANS

In this section we describe the algorithm for obtaining a plan of the execution. Different plans may be obtaind according to the rule by which the subsets E_j^v are chosen (see step 15).

At the moment j = 0 the sets C_j, C_j^{new}, E_j, E_j^{new}, F_j, and F_j^{new} are initialized to the set V_S of starting nodes. The set N_j is initialized to the set $V - V_S$. All the other sets are empty.

The plans for immediate and for lazy execution are computed by choosing $E_j^v := E_j - E_j^*$ and $E_j^v := \emptyset$, respectively, on each step (15). The plan for the heuristic execution is obtaind by choosing the sets E_j^v on each step (15) according to the sequence described on Fig.4.

The algorithm is implemented on LSI-11/23 computer under RT-11 operating system and tested on several program graphs. The Fig.7 describes an analysed program graph and the resulting time control vectors obtained.

(1) Evaluate $l(m,n)$ for all nodes m,n by computing the transitive closure of the program graph ;

(2) initialize sets ; $j := 0$;

(3) <u>while</u> $D_j \neq V$ <u>do</u>

<u>begin</u>

(4) $j := j + 1$;

(5) $F_j^* := \{ n \mid n$ finished execution at the moment $j \}$

(6) $C_j^{old} := C_{j-1} - F_{j-1}^{new}$;

(7) $C_j^{new} := \{ n \mid n$ received at the moment j its first input operand $\}$

(8) $C_j := C_j^{old} \cup C_j^{new}$;

(9) $E_j^{old} := E_{j-1} - F_{j-1}^{new}$;

(10) $E_j^{new} := \{ n \mid n$ received at the moment j its last input operand $\}$

(11) $E_j := E_j^{old} \cup E_j^{new}$;

(12) $F_j^{old} := F_{j-1} - F_j^*$;

(13) $l'(F_j^{old}, V_F) := \max l'(n,V_F)$, where $n \in F_j^{old}$;

(14) $E_j^* := \{ n \in E_j \mid l(n,V_F) = l(E_j,V_F) \geq l'(F_j^{old},V_F) \} \cup (E_j \cap V_F)$;

(15) Choose a subset E_j^v of the set $E_j - E_j^*$;

(16) $F_j^{new} := E_j^* \cup E_j^v$;

(17) $F_j := F_j^{old} \cup F_j^{new}$;

(18) $D_j := D_{j-1} \cup F_j^*$;

(19) $N_j := V - (C_j \cup F_j \cup D_j)$;

(20) Adjust the components of time control vectors for the nodes in C_j^{new}, E_j^{new} and F_j^{new} ;

<u>end</u> ;

Fig.5: Algorithm for obtaining execution plans of a program graph.

9. CONCLUSION

What is the time comlexity of the algorithm on Fig.5? The time complexity of the step (1) is $O(|V|^3)$, since this is the time complexity of the computing the longest paths between all nodes in a graph [Law76]. The time complexity of the steps (2) to (20) is $O(|V|^2)$, since the loop (3) is entered $O(|V|)$ times, each of the steps (4) to (20) having time complexity $O(|V|)$. Consequently, the overall time complexity of the algorithm on Fig.5 is is $O(|V|^3)$.

The algorithm was tested on 400 randomly generated program graphs with at most 64 nodes. In only 0.5% of analysed graphs $\mu_{heu} > \beta$ was obtained. In all other cases μ_{heu} was less than or equal to β. Heuristic plan improved both immediate and lazy execution in 50.25% of cases. Furthermore, many (55.75%) of heuristic executions were also proved to be optimal, since the associated numbers of the processing elements needed equaled the values α, as was the case on the Fig.7. Note, that this number is pesimistic since there were very probably graphs that could not be executed using only α processing elements. The results are depicted on Fig.6.

It is to point out that the algorithm on Fig.5 can be used for minimization of some other resources such as the number of primary memory locations needed .

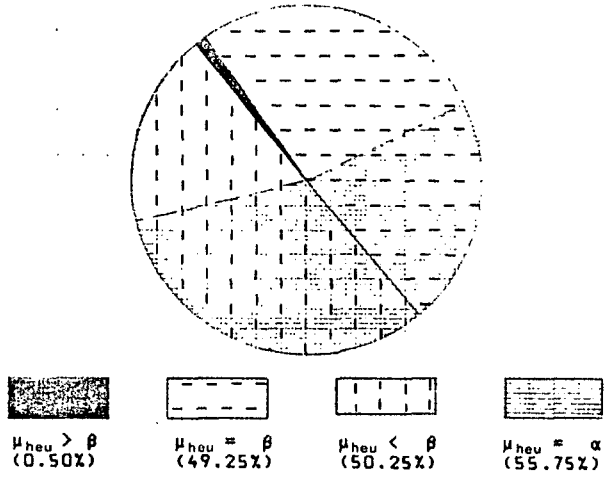
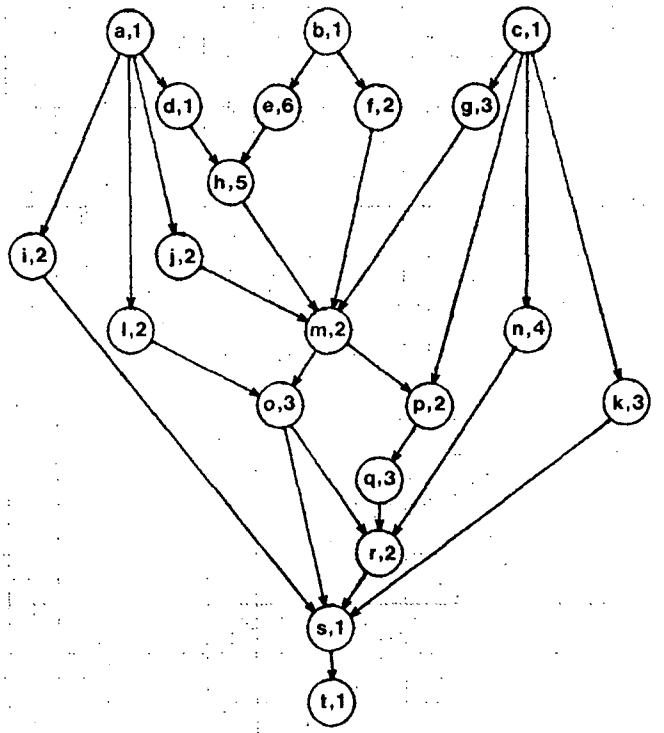


Fig.6: The behavior of the heuristic plan.

v	$\tau^{imm}_{F,v}$	$\tau^{lazy}_{F,v}$	$\tau^{heu}_{F,v}$
a	(0, 0, 0)	(0, 0, 0)	(0, 0. 0)
b	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
c	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
d	(1, 1, 1)	(1, 1, 6)	(1, 1, 1)
e	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
f	(1, 1, 1)	(1, 1,10)	(1, 1, 4)
g	(1, 1, 1)	(1, 1, 9)	(1, 1, 1)
h	(2, 7, 7)	(7, 7, 7)	(2, 7, 7)
i	(1, 1, 1)	(1, 1,19)	(1, 1, 9)
j	(1, 1, 1)	(1, 1,10)	(1, 1, 2)
k	(1, 1, 1)	(1, 1,18)	(1, 1, 6)
l	(1, 1, 1)	(1, 1,14)	(1, 1, 4)
m	(3,12,12)	(12,12,12)	(4,12,12)
n	(1, 1, 1)	(1, 1,15)	(1, 1, 6)
o	(3,14,14)	(14,16,16)	(6,14,14)
p	(1,14,14)	(1,14,14)	(1,14,14)
q	(16,16,16)	(16,16,16)	(16,16,16)
r	(5,19,19)	(19,19,19)	(10,19,19)
s	(3,21,21)	(19,21,21)	(9,21,21)
t	(22,22,22)	(22,22,22)	(22,22,22)



$\mu_{imm} = 9$ $\mu_{lazy} = 4$ $\mu_{heu} = 3$

$|\pi| = 3$ $\zeta = 1$ $\alpha = 3$

In this case the heuristic plan is also optimal!

Fig.7: Plans for immediate, lazy and heuristic execution of a given program graph.

10. LITERATURE

[COM82] COMPUTER, Soecial Issue On Dataflow Systems, Vol.15, No.2, Feb.1982

[KFGS84] Kapauan A., J.T.Field, D.B.Gannon, L.Snyder: 'The Pringle Parallel Computer', Proc. 11th Int'l Symp. Comp. Arch., IEEE Press, New York, 1984, pp.12-20

[Law76] Lawler E.: Combinatorial Optimization: Networks and Matroids, Holt, Rinehart&Winston, New York, 1976

[RoŠ86] Robič B., J.Šilc : 'Analysis of Static Data Flow Program Graphs', to be published in Elektrotehniški vestnik, (in Slovene)

[TJS83] Tokoro M., J.R.Jagannathan, H.Sunahara: 'On the Working Set Concept for Data-flow Machines', Proc. 10th Int'l Symp. Computer Architecture, IEEE Press, New York, 1983, pp.90-97

[ŠiR85] Šilc J., B.Robič : 'Basic Principles of Data Flow Systems', Informatica 2/85, pp.10-15 (in Slovene)