ALLOCATION FOR THE SANDAC MULTIPROCESSOR SYSTEM

T. M. Ravi
M. D. Ercegovac

February 1986
CSD-860059

ABSTRACT

In this report we describe an algorithm for the static allocation of tasks in a general Dataflow Multiprocessor and the SANDAC IV System in particular. Initially a model of execution and the underlying assumptions about the architecture are outlined. We then discuss a Graph Reduction algorithm for preprocessing the computation graph. The Graph Reduction algorithm reduces a fine grain graph to an optimal grain graph. The heuristic allocation algorithm is presented and is based on giving precedence to critical paths and minimizing the communication time between tasks. The performance of the algorithm is then analyzed and the effect of varying parameters is studied. Subsequently we propose an alternative variation with better characteristics.

In the appendix details of the software implementation and its use is demonstrated.

ALLOCATION FOR THE SANDAC MULTIPROCESSOR SYSTEM

T. M. Ravi and M. D. Ercegovac

UCLA Computer Science Department

Introduction

In this report we discuss the allocation of tasks in the SANDAC IV system ([BORG 83]). Initially we outline the model of execution and the underlying assumptions. We then discuss a graph reduction algorithm for preprocessing the computation graph, which is particularly necessary if the graph is very fine grain. The allocation algorithm is presented along with performance curves for different graphs. In the appendix, details of the software implementation and its use is discussed.

Model of Computation

The program is represented by a data flow graph ([DENN 80]), with nodes representing tasks and arcs representing precedence relationships between tasks. The partial ordering of the tasks necessary for correct execution is captured by the dependencies between these tasks. The nodes have a single point of entry and a single point of exit, i.e., a task can begin execution only when all its inputs (arguments) have arrived, and can deliver each of its results to destination tasks only after the execution of the task is completed. Likewise, the graph has a single entry node and a single exit

* This work has been supported in part by the Contract No. 25-3074 from the Sandia National Laboratories "Multiprocessor System Evaluation and Programming Environment"

node.

To represent control structures such as conditionals and loops in data flow graphs we introduce two special nodes (Figure 1). The "OR" node has three input arcs and one result arc. One of the arguments is boolean, and depending on its value, a token from one of its arcs (true or false arc) is processed and placed on the result arc. This special node is unlike other nodes which require all inputs to be present before the node can be activated.

The "SW" node has two input arcs, one being boolean; and two result arcs (True and False). Depending on the boolean value the result token is put on one of the result arcs. The "SW" and "OR" are in the same flavor as the Switch and Merge actors discussed by [DENN 80]. The "SW" operator on firing will output a token on either of its output arcs and the "OR" will fire when a token is present on any one of its input arcs.

Our present implementation of the allocation algorithm is for directed graphs without loops. Loops implemented by "SW" and "OR" operators could be handled by applying our algorithm in an hierarchical manner.

It is assumed that the execution time (t_p) of each node (tasks) is known apriori. There is a communication time (t_c) associated with each arc in the graph, whose value depends on the size of data communicated. Furthermore, the communication time can take on a lower value - local communication time (t_{cl}), or a higher value - bus communication time (t_{cb}). Bus communication time is chosen if results from one task have to be sent to another task in a different processor. Local communication time is chosen when tasks reside in the same processor. One point to note is that the processors are busy during communication and will not become available until all the results are sent to their destinations. Results are sent out sequentially, due to limitations imposed by the communication mechanism, and hence the total communication time (t_c) is the sum of individual communication times of each result.

A task once started is not interrupted and will run till completion. A task can be activated only when all its arguments have arrived.

The objective is to allocate the tasks to a multiprocessor (given n processors), in order to obtain minimum execution times.

Graph Reduction

To reduce the complexity of the allocation process and to utilize the parallelism efficiently, we can reduce ([GAUD 84] & [ERCE 84]) the original graph into a larger grain task graph. By applying a set of rules, subgraphs in the data flow graph are replaced by a single node. The criterion for lumping together instructions into a single task is to minimize the response time for the subgraph under consideration.

When the delay incurred due to interprocessor communication and activation exceeds the gain in time due to concurrent execution, it is no longer justifiable to distribute the nodes over several processors. When the response time of a subgraph executed sequentially in a single processor is less than or equal to the response time when executed concurrently, then the subgraph is reduced to a single node and is executed sequentially.

The condition ([RAVI 86]) for combining a node with its arguments is:

$$\sum_{i=1}^{narg} t_{parg} \leq \max_{i}(t_{parg} + t_{carg})$$

where t_{parg} is the processing time of the argument node

t_{carg} is the communication time of the argument node and

narg is the number of arguments.

If this condition is satisfied then the node and its argument nodes are lumped together into a single node.

This step is illustrated in Figure 2. Figure 2a is a subgraph where the nodes are separated in order to take advantage of the parallelism, while in Figure 2b the nodes A, B and C have been lumped together into a single node. In the subgraph of Figure 2a, node D can execute only after the results from node A and B and C have arrived. If nodes A, B and C are activated at the same time, then the result from nodes A and B will arrive after 5 cycles and the result from node C will arrive after 8 cycles. Hence node D is activated only after 8 cycles. In the sequential case the result from nodes A, B and C are available after 6 cycles, as we do not have to communicate between different processors. In this case the subgraph of Figure 2a can be reduced to Figure 2b.

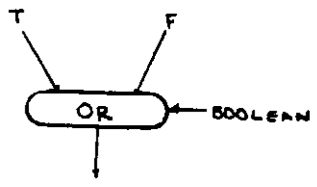
```
Procedure main (G:typegraph);
{This procedure increases the grain size of the data flow graph (G).
 Starting at the root , nodes are combined with its arguments.}

begin
 UPREDUCTION(Root(G));
end;{main}


Procedure UPREDUCTION (i:typenode);
{This recursive procedure lumps a node and its arguments
 together, based on criterion depending on the the processing
 time and communication time. Each node has the the fields
 argument (arg), no. of arguments (narg), code (funct),
 processing time (proctime) and communication time
 (commtime).}

begin
 with node[i] do
  if narg > 0 then begin
  {test condition}
   seqtime:=0; partime:=0;
   for k:=1 to narg begin
     seqtime:=seqtime + node[arg[i]].proctime;
     if (node[arg[k]].proctime + node[arg[k]].commtime)
             > partime then
       partime := node[arg[k]].proctime + node[arg[i]].commtime;
   end;
   if ((partime -seqtime ≤ 0) or (if any arg has > than one result)) then
     {condition for reduction of parallelism is not true}
     for k:=1 to narg do UPREDUCTION(k);
   else begin
     {condition is false}
     copy the code in each of the arguments to node[i].funct
     node[i].proctime := seqtime;
     node[i].narg := sum of the narg of each of the arguments
                  of node[i]
      arguments of new node := arguments of all nodes combined
                with node[i]
     remove the old argument nodes from graph
     UPREDUCTION(i);
    end;
 end;{UPREDUCTION}
```

Figure 3 : Upreduction Algorithm



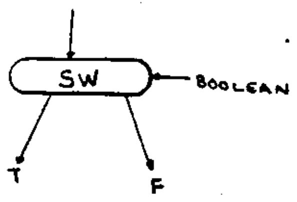


Figure 1 : OR and SW operators

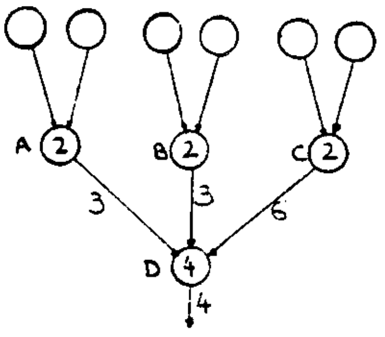


Figure 2a : Fine Grain Graph

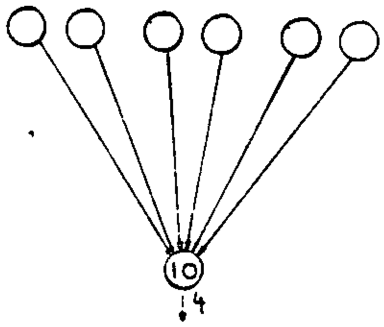


Figure 2b : Lumped Graph

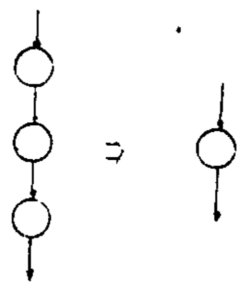


Figure 4 : Reduction of Sequential Nodes

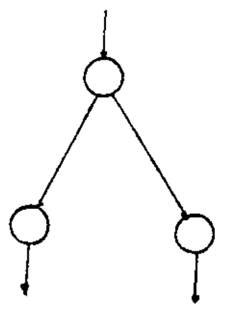


Figure 5 : Node with many results

The "upreduction" algorithm (Figure 3) spans the graph, testing criterion for reduction, in O(n) time. It combines a node with its arguments whenever the reduction criterion is met.

Note that sequential nodes which have single arguments and single results are combined together into a single node (Figure 4). Execution of each of the sequential nodes in a different processor leads to unnecessary overhead.

However, when a node has more than one result which goes to different nodes, then it can not be combined by the "upreduction" algorithm. In order to reduce these subgraphs (Figure 5), a "downreduction" algorithm has to be applied with the entry node as a parameter. It combines a node and its results based on the processing time and communication time criterion into a single node. The algorithm is similar to the "upreduction" algorithm.

The graphs of Figure 6a,6b & 6c illustrate the Graph Reduction algorithm, with an example of an iteration consisting of 30 nodes (Figure 6a), which also has a conditional statement in it. After a single pass of the reduction algorithm, i.e., combination of a node and its arguments, we obtain a graph with 19 nodes (Figure 6b). After another pass of the reduction algorithm , i.e., combination of a node and its results, we obtain the final reduced graph consisting of 13 nodes (Figure 6c). We are now ready to allocate this graph to the processors.

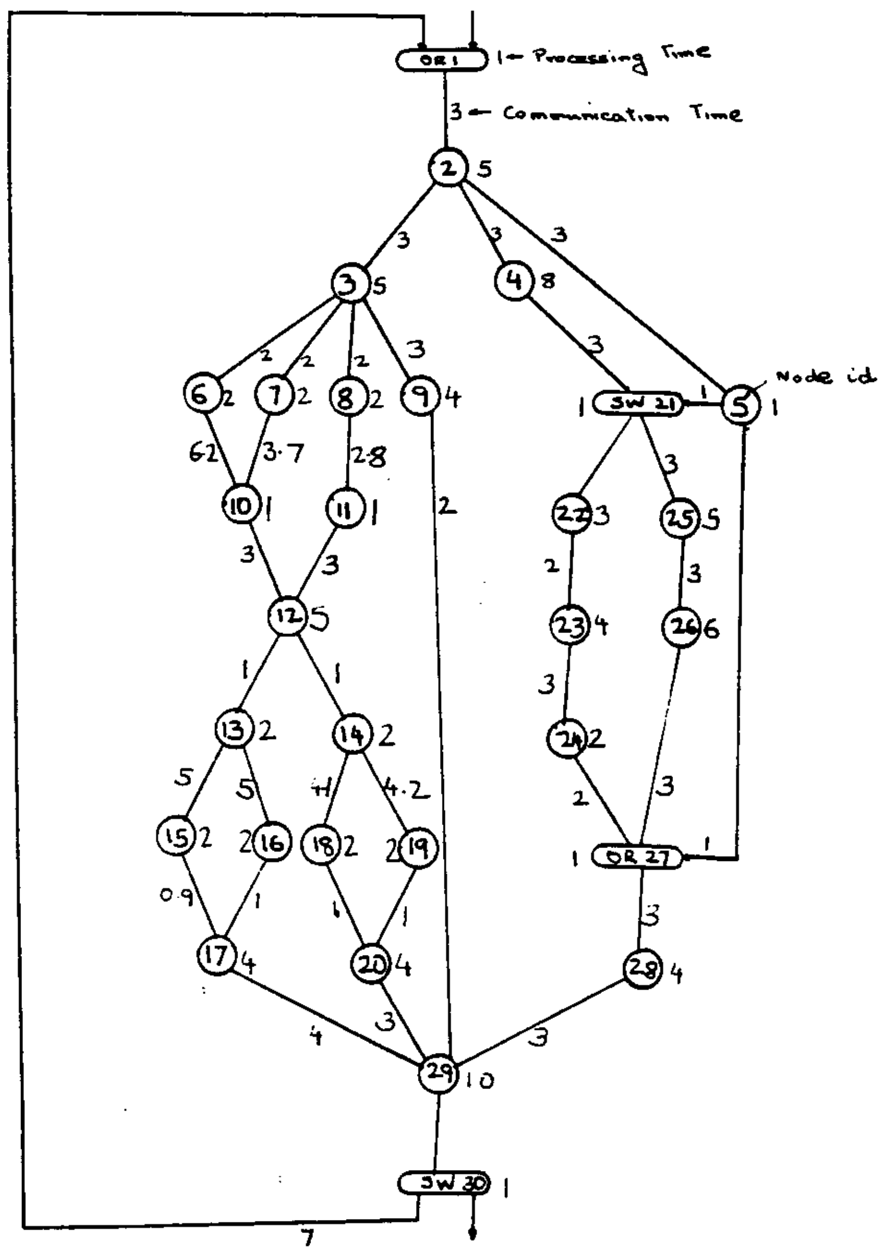


Figure 6a : Initial Data Flow Graph (30 nodes)

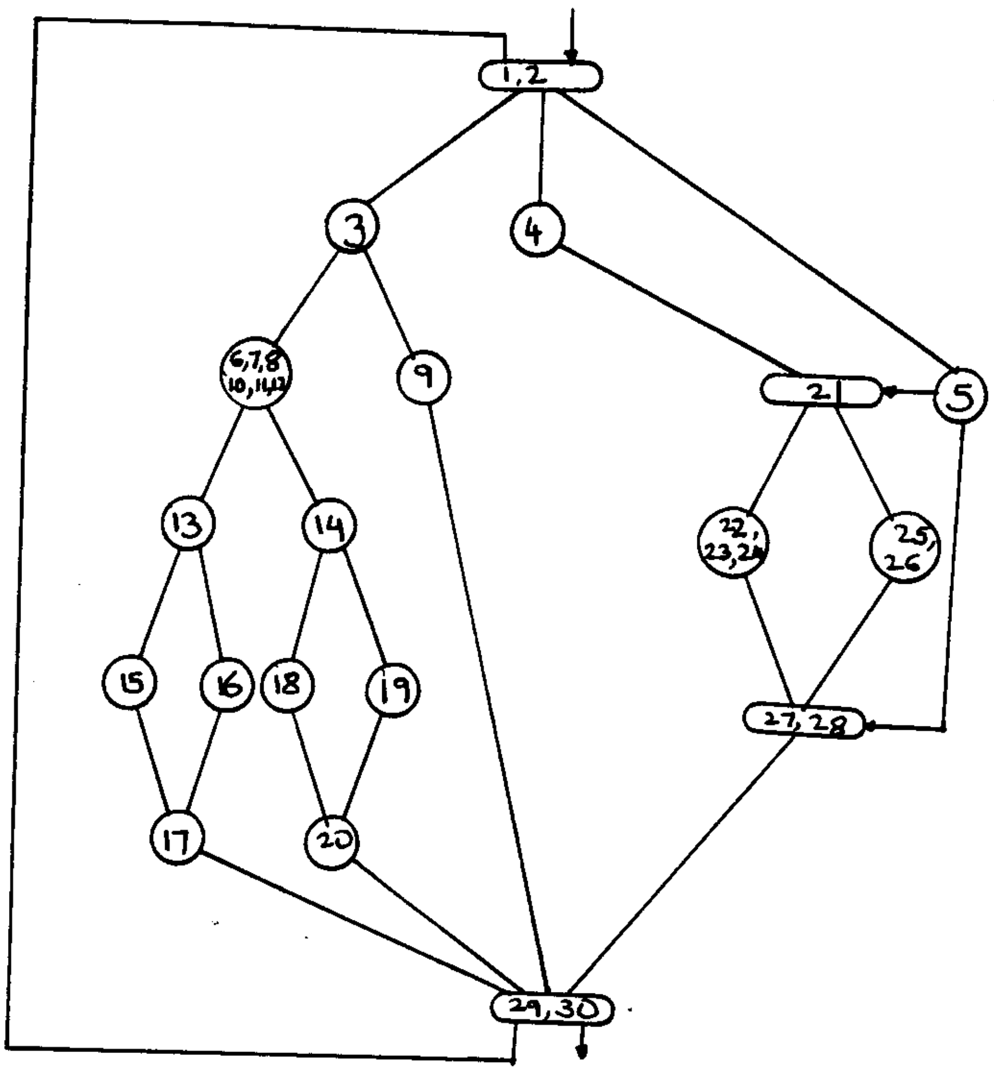


Figure 6b : Intermediate Graph After Upward Reduction (19 nodes)

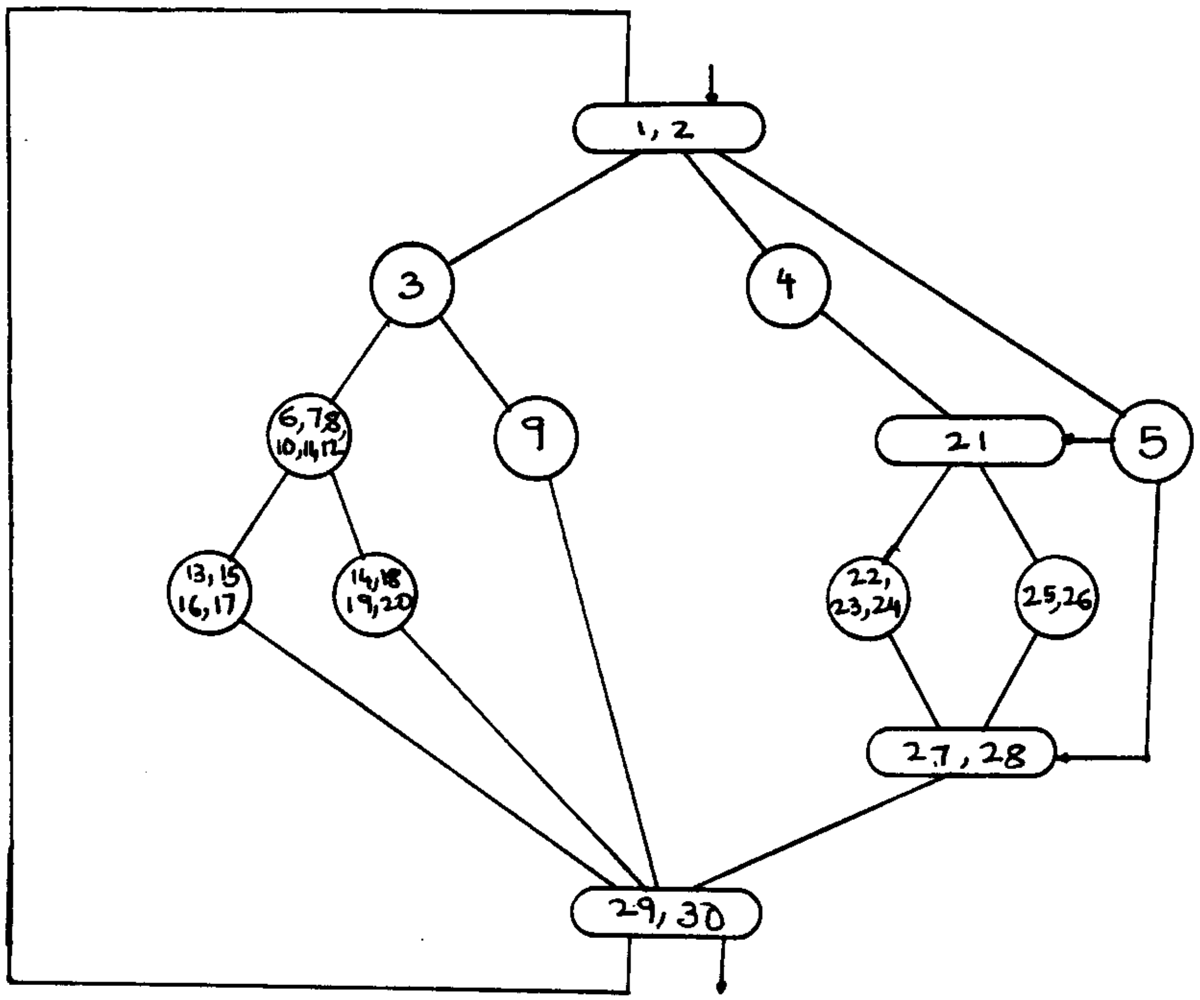


Figure 6c : Final Graph After Downward Reduction (13 nodes)

Task Allocation

The heuristic allocation algorithm minimizes response time based on two principles :

1) Precedence to critical tasks

2) Minimizing communication time between tasks

An allocation algorithm based on the first principle of critical path scheduling, when the tasks only have processing times associated with them is discussed in [KOHL 75]. The second principle of minimizing communication time provides a criterion for selecting a task for allocation when several candidates are available. It enables us to allocate predecessor-successor tasks to the same processor, thus incurring the lower local communication time.

The difficulty in applying the critical path algorithm to this problem is that timing parameters associated with the graph cannot be fixed until the allocation is itself complete. This is because the decision on whether to choose local or bus communication time for an arc depends on where the successor task will be allocated. This leads to two specific problems. First, critical paths which are the longest paths in the graph cannot be precisely determined. Second, when a task is allocated to a processor, we cannot determine exactly when the task will complete, because it is not known at that stage in the algorithm as to where the successor tasks are going to be allocated in order to choose the right communication time. In our algorithm we show how these two problems can be handled.

The Algorithm

Consider a graph with tasks T_1, T_2 .... T_k, to be executed on n processors P_1, P_2 .... P_n. Two lists are constructed - Processor list (L_p) and Task list (L_t). The processor list, at any stage of the algorithm, contains the processors listed in increasing order of busy times, i.e., the time up to which they are busy. The processor on the top of the list is the one which will become free next. Initially, the processors are in random order in the list, as they are free. The task list is generated based on critical path lengths. The critical path length (CP(T_i)) of a task T_i, is defined to be the length of the longest path from the exit node to T_i. To calculate the critical paths, we assume that the value of the communication time taken for each arc is the higher bus time. The critical paths of nodes in a graph are calculated starting from the exit node. The critical path of the exit node is equal to t_p+t_c, where t_c is the sum of the bus communication times of all the results. The critical path of any other node in the graph is equal to the maximum critical path of result nodes + t_p + t_c, where again t_c is the sum of the bus communication times of all the results. The task list (L_t) is generated by sorting the tasks in decreasing order of their critical paths. At any stage of the algorithm, the list contains tasks yet to be allocated.

At any time we choose the top processor from the processor list (L_p), which is the first to become idle. The task list is then scanned till we can choose the first candidate for execution in the processor. Any other task on the list which can be executed, and is within a deviation of Δ from the critical path of the first candidate, is also chosen as a candidate. A task can be a candidate only if at the time when the processor becomes free all its arguments have arrived, i.e., all its predecessors have completed execution.

Now we choose the task among the candidates to be assigned to the processor. Of all the candidate tasks, we choose the task which when allocated to the processor gives the maximum saving in communication time. A saving in communication time is made if the predecessor tasks are assigned to the same processor. The saving is the sum of the difference of the bus communication time and local communication time for each direct predecessor assigned to the same processor.

The chosen task is assigned to a processor, but the question that arises is - What will the duration of the execution of this task be ? This would be $t_p + t_c$, but we don't know whether to take the local or bus communication time for the results of the task, as the successor tasks have not yet been allocated.

The solution to this problem is to associate communication times with arguments instead of results. Thus, when a task is allocated, the location of its predecessor is known. In our model the communication of the results is the responsibility of the task, and to take care of this we reverse the graph. The direction of the arcs in the graph is reversed before the calculation of critical paths and the generation of lists. On starting with the reversed graph, the schedule obtained can be reversed to obtain a regular allocation. By reversing the graph, the communication time of the arcs is associated with arguments to tasks and not results.

After the task has been assigned to the processor, the busy time of the processor is updated. The task is removed from the task list (L_t) and the processor is reinserted in the appropriate position in the processor list (L_p), which is ordered according to increasing busy time.

If no task can be assigned to the processor (P_1), then we have to move to the time of the next event and try again. The processor list is scanned; and the first processor (P_2) with busy time greater than the busy time of this processor (P_1) is placed on the top of the list. Processor P_1 and any other processors with busy time equal to that of P_1 are updated with busy time equal to the busy time of P_2. In this way idle times are caused in processors when no tasks are ready.

This process of allocating each task to a processor continues till the task list is exhausted. The allocation algorithm is given in Figure 7.

```
Procedure ALLOCATE(G);

{ This procedure allocates the tasks T[1], T[2] .... T[k] of the computation
  graph G to the n processors P[1], P[2] .... P[n]}

var
   $L_p$ : List of processors;
   $L_t$ : List of tasks yet to be allocated;
   listsize : No. of tasks in $L_t$;
   candidate: List of tasks that may be allocated to top processor in $L_p$;

begin
   Reverse graph G to $G'$ by reversing direction of all arcs in G;
   EVALCP(Root($G'$));

   {initializing lists}
   sort tasks by T[i].CP
   $L_t$[1]:=p; $L_t$[2]:=q; ....... $L_t$[3]:=s
    where T[p].CP > T[q].CP .... > T[s].CP;
   listsize :=k;
   $L_p$[1]:=1; $L_p$[2]:=2; ...... $L_p$[n]:=n; {any random order}

   while listsize > 0 do begin
      SELCANDIDATES(candidate,nocandidates,$\Delta$,listsize);
      if nocandidates > 0 then begin
         SELTASK(candidate,nocandidates,chosen-task,saving);
         remove chosen-task from $L_t$;
         update list;
         listsize:=listsize-1;
         P[$L_p$[1]].busytime := P[$L_p$[1]].busytime +
           T[chosen-task].$t_p$ + $\sum$ T[chosen-task].$t_{cb}$ - saving;
         T[chosen-task].completion-time := P[$L_p$[1]].busytime;

         Sort processors in $L_p$ so that
           P[$L_p$[1]].busytime < P[$L_p$[2]].busytime < P[$L_p$[n]].busytime;
      end else begin {if nocandidates = 0}
         Go down list $L_p$ starting at $L_p$[1] till an entry r with
            P[$L_p$[r]].busytime > P[$L_p$[1]].busytime is found;

         Place processor found at $L_p$[r] at $L_p$[1] i.e., at the top of the list;
         update list;
         for j:=2 to r do $L_p$[j].busytime := $L_p$[1].busytime;
      end;
   end;
end; {ALLOCATE}
```

Figure 7a : Allocation Algorithm

```
procedure EVALCP(r:typetask);

{ This procedure computes the critical paths T[i].CP of each task T[i]}

begin
  if T[r].narg = 0 then T[r].CP := T[r].$t_p$ + $\sum$T[r].$t_{cb}$
  (where $\sum$T[r].$t_{cb}$ is the sum of bus comm. times of all results)
  else begin
    maxrescp := $\underset{j \varepsilon \text{ res tasks of } T[r]}{Maximum} T[j].CP$

    T[r].CP := T[r].$t_p$ + $\sum$T[r].$t_{cb}$ + maxrescp;
  end;
  for each argument task (p) of task (r) do EVALCP(p);
end; {EVALCP}
```

```
procedure SELTASK (candidate,nocandidates,var chosen-task,var saving);

{This procedure chooses a task among the candidates which will locally be most
 beneficial}

begin
  for i:=1 to nocandidates do begin
    candidate[i].saving:=0;
    for j:=1 to T[candidate[i]].narg do begin
      if argument task (p) has been allocated to the same processor $L_p$ [1] then
        candidate[i].saving:=candidate[i].saving +
              $(t_{cb} - t_{cl})_{\text{arc } from\ arg.\ to\ candidate[i]\ task}$
    end;
  end;
  saving := $\underset{i =< nocandidates}{Maximum}$ (candidate[i].saving);
  chosen-task :=r; {value of i which gives above maximum}
end; {SELTASK}
```

Figure 7b : Allocation Algorithm (Critical Path Calcuiation and Task Selection)

```
procedure SELCANDIDATES(var candidate, var nocandidates,$\Delta$,listsize);

{This procedure selects tasks which can be executed next on the processor $L_p$ [1]}

begin
  i:=1; nocandidates:=0;
  while ((i =< listsize) and (nocandidates=0)) do begin
    if ((T[k].completion-time =< P[$L_p$ [1]].busytime for all argument
        tasks (k) of task $L_t$ [i])
      or (T[$L_t$ [i]].narg = 0)) then begin
        candidate[1] :=$L_t$ [i];
        nocandidates:=1;
    end else i:= i+1;
  end;

  if nocandidates > 0 then begin
    i:=i+1;
    limit:=T[candidate[1]].CP - $\Delta$;
    while i =< listsize do begin
      if ((T[k].completion-time =< P[$L_p$ [1]].busytime for all
          argument tasks (k) of task $L_t$ [i])
        or (T[$L_t$ [i]].narg = 0)) then begin
          candidate[1] :=$L_t$ [i];
          nocandidates:=1;
      end;
      i:= i+1;
    end;
  end;
end; {SELCANDIDATES}
```

Figure 7c : Allocation Algorithm (Selection of Candidate Tasks)

Performance

To study the performance of the algorithm, several program graphs were allocated and statistics collected. The effect of changing the parameter Δ, which is the deviation in critical path for the choice of candidates, and the behavior of the algorithm for different ratios of processing time and communication time, were studied.

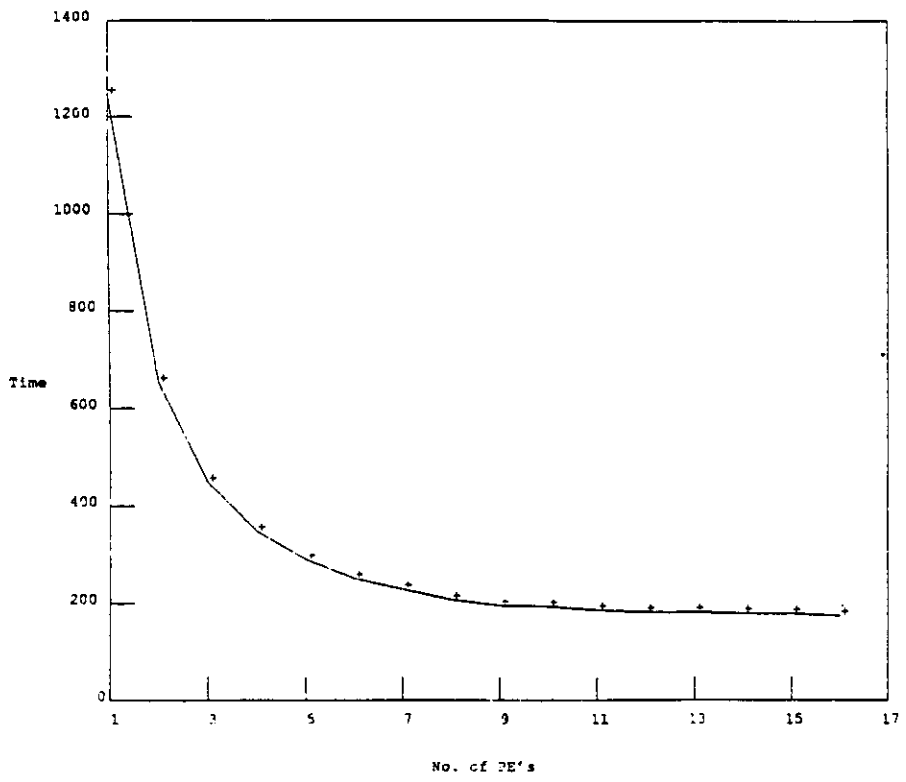


Figure 8 : Response Time (T) Vs. No. of Processors (N)

We first examine the speedup achieved by using multiprocessors. Figure 8 shows the variation of response time (T) with the number of processors (N) for a graph (Figure 9) containing 123 nodes. The processing time of each node is 20, the local communication time 0.1, the bus communication time 1 and the deviation (Δ) 1 unit of time. We observe that initially when the amount of concurrency exceeds the number of processors available, the response time falls rapidly with the increase in the number of processors. Figure 10 illustrates the speedup (T[1]/T[i]) of the multiprocessor system over a single processor. With a multiprocessor system

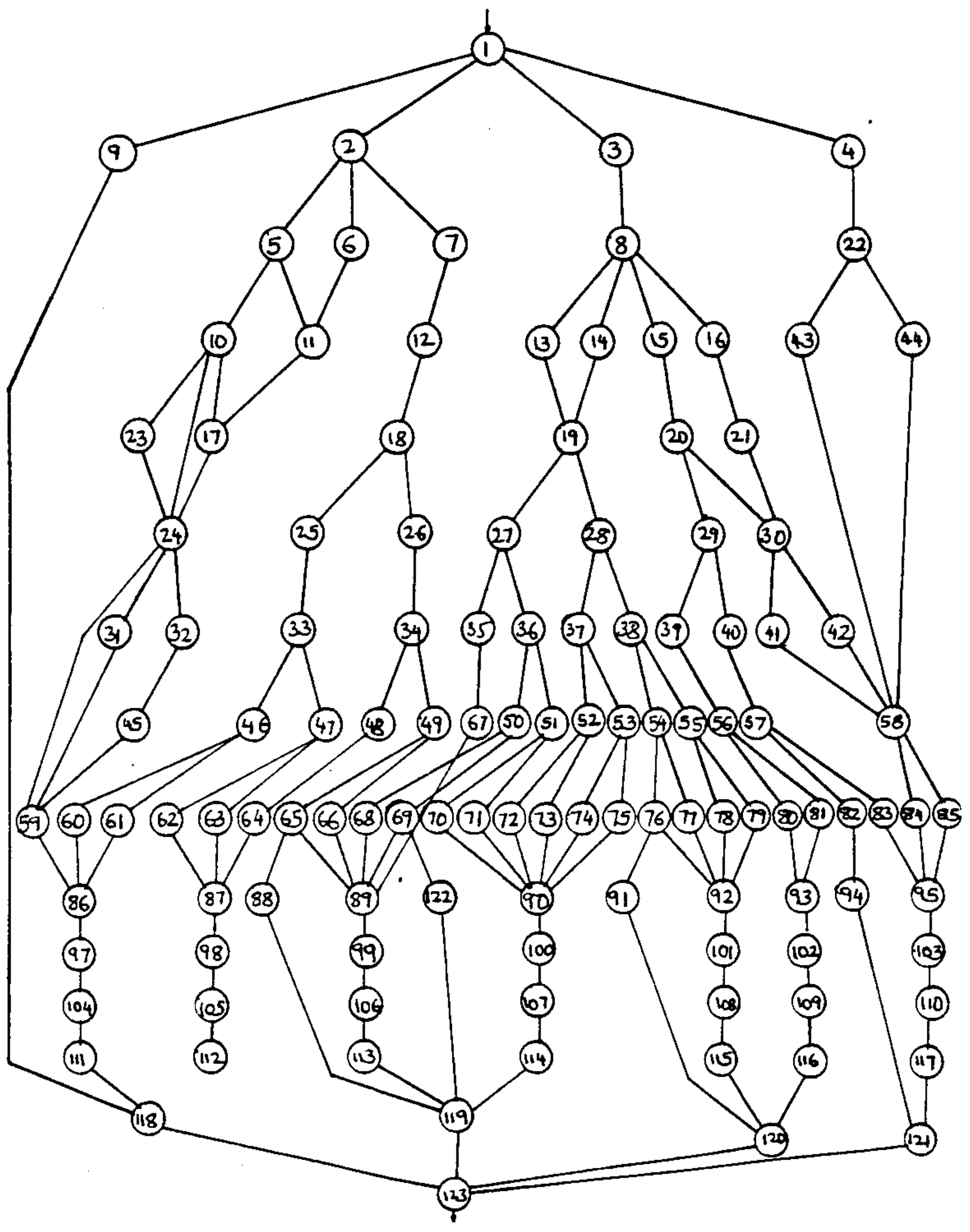


Figure 9 : Graph With 123 Nodes

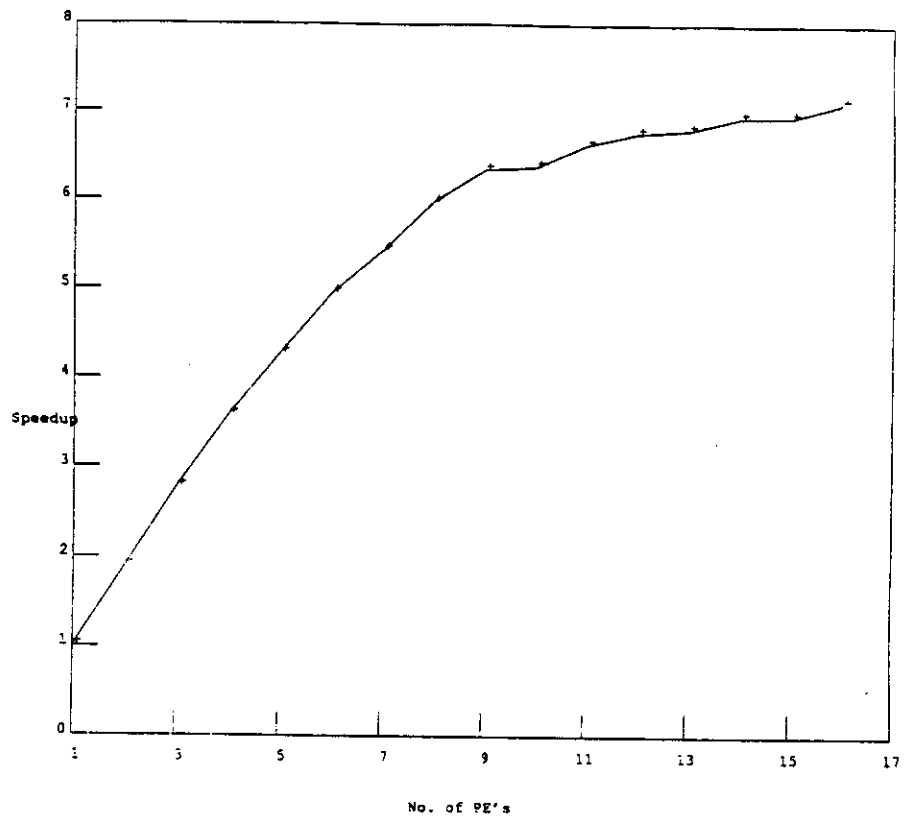


Figure 10 : Speedup Vs. No. of Processors

consisting of 8 processors, the speedup over the uniprocessor is 6. Initially, when the number of processors is increased the speedup is almost linear, but as the amount of concurrency is exhausted the curve saturates. Figure 11 demonstrates the efficiency (Speedup/N) of the processors in the multiprocessor system. The fall in efficiency is attributed to the dependencies in the graph which force idle times in some processors when very few tasks can be activated.

The algorithm has two driving principles - Precedence to critical tasks (critical path scheduling) and the minimization of communication time between tasks. Figure

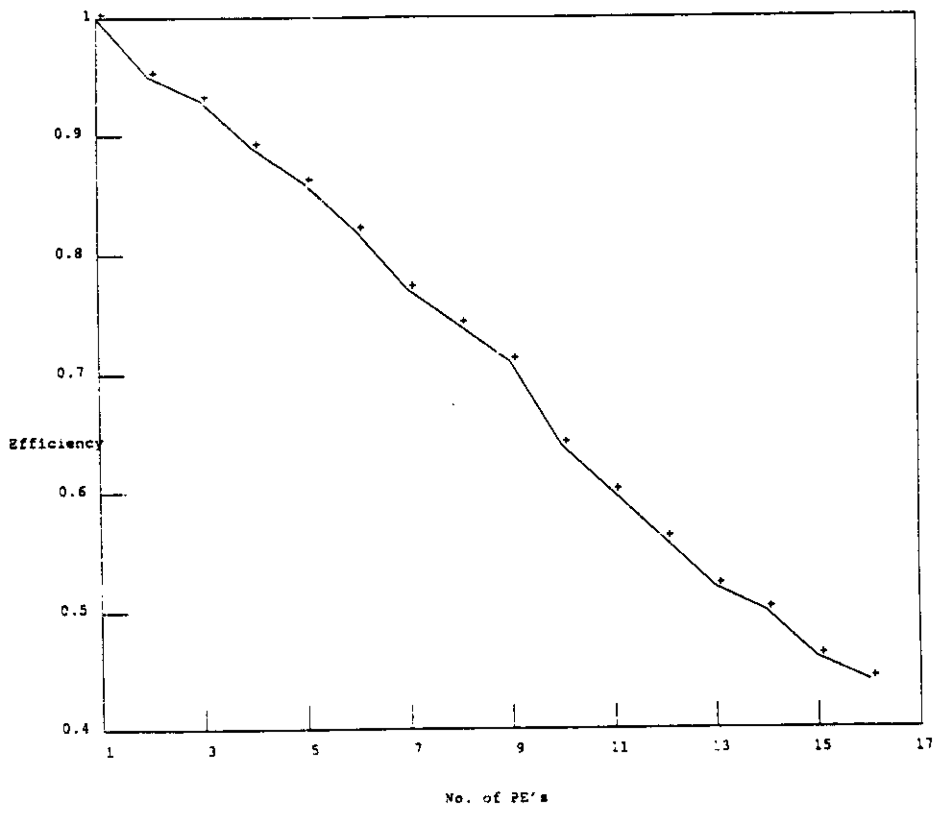


Figure 11 : Efficiency Vs. No. of Processors

12 shows the performance when only critical path scheduling is enforced. The example is of a sort-merge graph (Figure 13) with 94 nodes, where the processing time of each node is 20 units, the local communication time is 0.1 units and the bus communication time is 5 units. The curve (a) shows the response time for a strict list schedule where no attempt is made to have predecessor-successor tasks cohabit in the same processor. Curve (b) uses our algorithm with a deviation (Δ) equal to 0.1, which is the local communication time. The deviation (Δ) is usually chosen to be a factor of the bus communication time. For two processors the difference in the response times is 15%, due to the large saving from the reduced interprocessor communication.

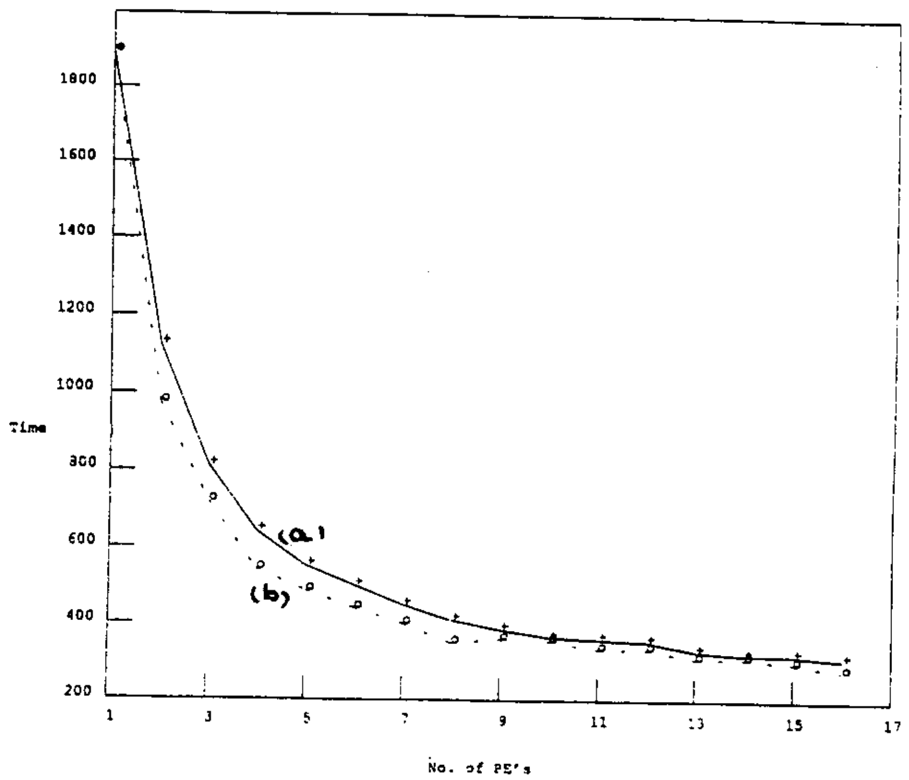


Figure 12 : Comparison of Performance with Critical Path List Schedule

When the deviation is very large, i.e., several orders of magnitude larger than the bus communication times, then the critical path list ordering is no longer operative. In Figure 14 we have a program graph with one dominant critical path and several non-critical tasks. When the deviation exceeds the length of the critical path, then at each stage the candidates for allocation to a processor are all the enabled tasks in the graph. In other words critical and non critical tasks are given equal chance for execution at any point. For two processors for the graph of Figure 14, with t_p =20, t_{cb} = 1 & t_{cl} = 0.1 we observe that the response time increases by 22% from zero

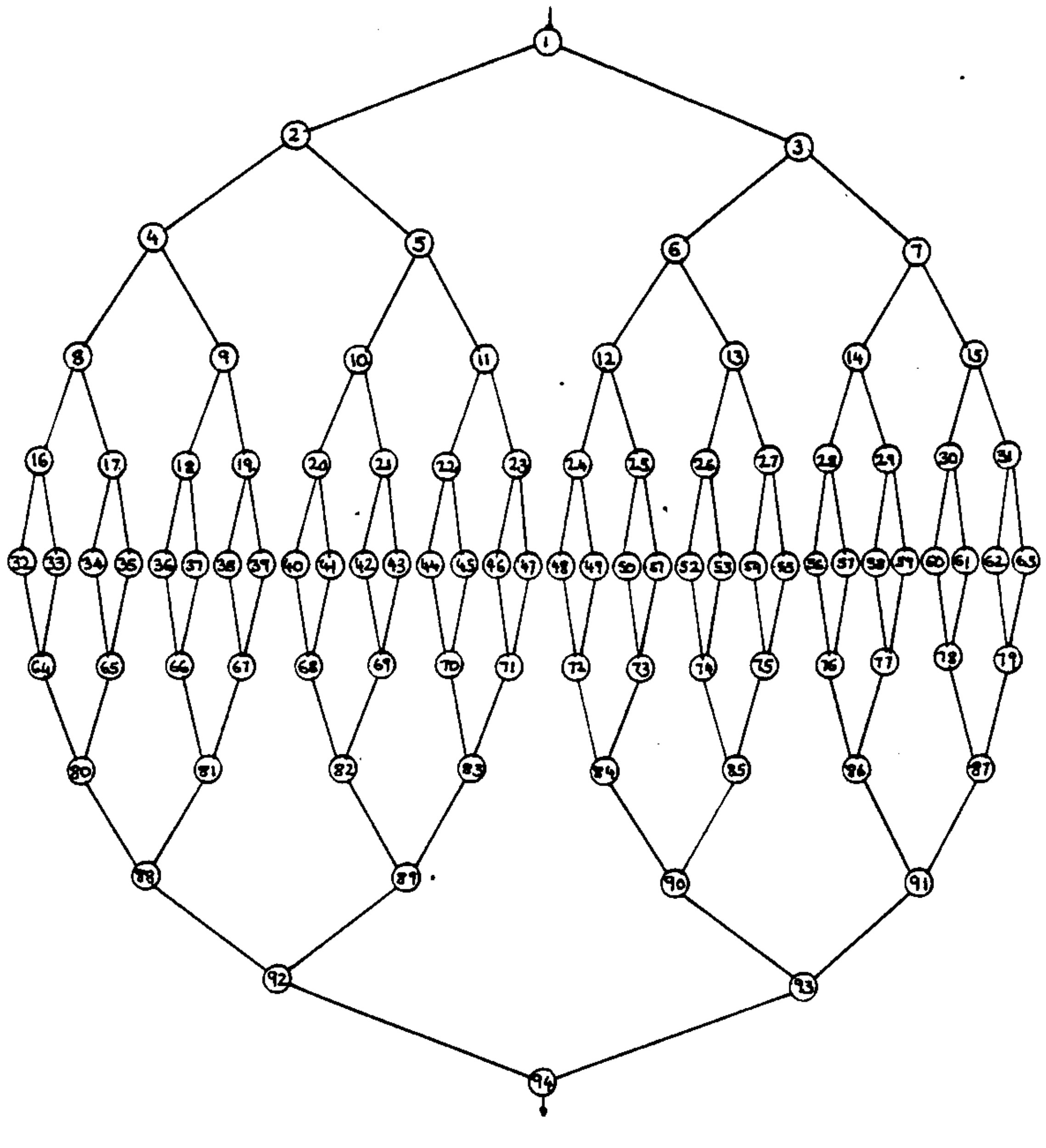


Figure 13 : Sort-Merge Graph

deviation response time, when the deviation is greater than the critical path.

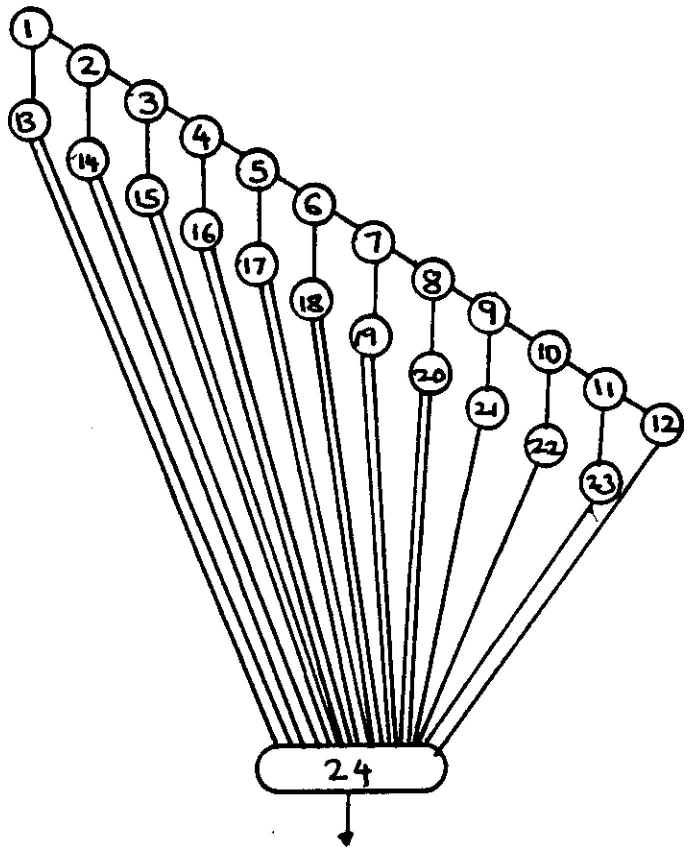


Figure 14 : Example with a Dominant Critical Path Schedule

A Variation to the Allocation Algorithm

One variation to the Allocation algorithm which we have considered is to evaluate critical paths based on the processing time alone. The motivation behind this variation (A_{CP}) to the algorithm is that here the communication times (bus or local communication times) will not influence the order of tasks in the critical path list. Our observation with the example (Figure 9) with 123 nodes shows that when the bus communication times are low then the difference between the response times from the two algorithms is insignificant. But as the bus communication time increases, the modified algorithm (A_{CP}) performs noticeably better.

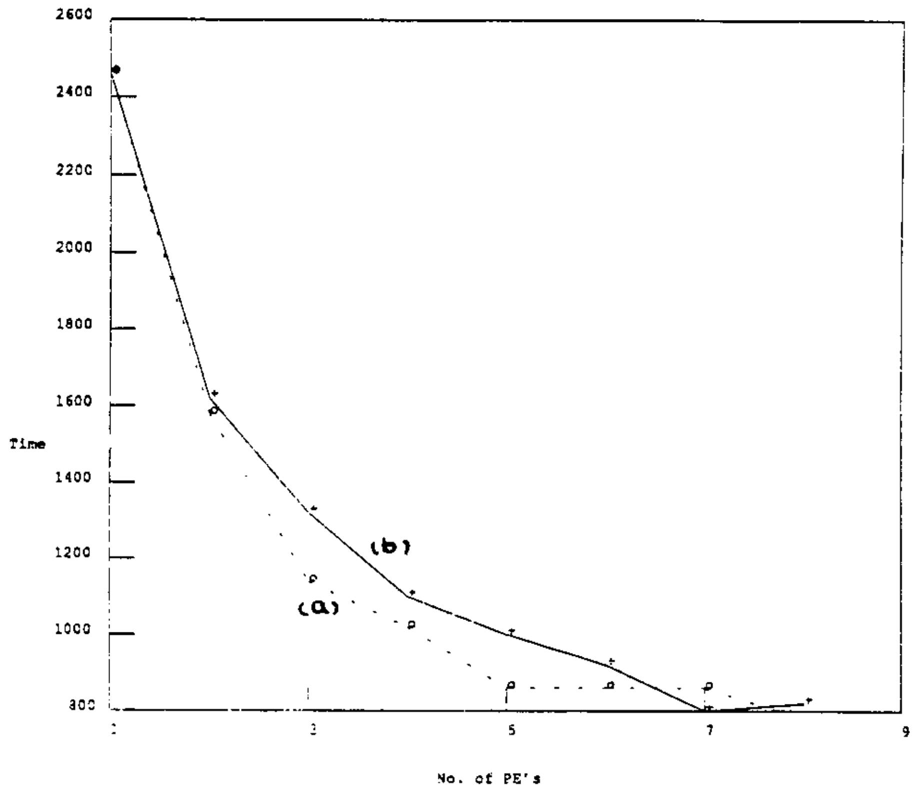


Figure 15 : Comparison of Performance with Modified Algorithm

In Figure 15, curve (a) indicates the response time of the modified algorithm (A_{CP}), while curve (b) is that of the original algorithm. In this example the bus time is equal to the processing time of the task, implying low grain parallelism (t_p =20, t_{cb} = 20, t_{cl} = 0 & Δ = 1) A reduction in response time of upto 16% (for 3 processors) indicates this variation is an improvement to the original algorithm.

Acknowledgements

Many of the ideas in this paper resulted from a series of discussions with Professor Tomas Lang and Professor Richard Muntz. We are grateful to them for their suggestions and constructive criticism of this work.

The Data Flow Model of the SANDAC Multiprocessor System was developed with the close cooperation of Mr. George Davidson. We are thankful to him for his close and constant support and for his active help in our understanding of the architecture.

Reference

[BORG 83] Borgman, C. R. and P. E. Pierce, "A Hardware/Software System for Advanced Development Guidance and Control Experiments," *AIAA Computers in Aerospace Conference*, AIAA-83-2416, Oct. 1983, Hartford, CT, pp. 377-384.

[DENN 80] Dennis, J. B., "Data Flow Supercomputer Languages," *Computer*, Nov. 1980, pp. 48-56.

[ERCE 84] Ercegovac, M. D., P. K. Chan and T. M. Ravi, "A Dataflow Multiprocessor Architecture for High Speed Simulation of Continous Systems," *Proc. International Workshop on High-Level Architecture, 1984*.

[GAUD 84] Gaudiot, J. L., and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proc. 4th International Conf. on Distributed Computing Systems*, 1984, pp. 2.9-2.17.

[KOHL 75] Kohler, Walter H., "A Preliminary Evaluation of Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, Vol. C-24, Dec. 1975, pp. 1235-1238.

[RAVI 86] Ravi, T. M., "Partitioning and Allocation of Functional Programs for Data Flow Processors," *M.S. Thesis*, UCLA Computer Science Department, Feb. 1986.

Appendix 1

The software tools for the allocation of tasks to the SANDAC IV architecture consists of two programs - allocation.p and reduction.p, implemented in Berkeley Pascal and given in Appendices 2 and 3 respectively. The input file to reduction.p is 'ingraph'. Appendix 1.1 shows the format of ingraph for the graph of Figure 5a. The main program allocation.p has input file 'outgraph2' if the original graph is to be allocated and 'outgraph4' (Appendix 1.2) if the reduced graph is to be allocated. Files 'outgraph2' and 'outgraph4' are output files from reduction.p. A session illustrating the execution of the programs is given below.

```
<1> reduction.out
Want to parameterize Communication  and Processing times (y or n) ?y
Processing time:10
Local Communication time:0.5
Bus Communication time:5
Pr.time =  1.00e+01
LocComm.time =  5.00e-01
BusComm.time =  5.00e+00
<2> allocation.out
Input is original graph {o} or reduced graph {r} r
Reading reduced graph
The critical path of the graph is    185.000
No. of processors =2
 Deviation in critical path for selecting candidates =0
 Deviation in critical path is =      0.00
The response time for         2 processors is    206.500
```

The output files of allocation.p are 'outstat' and 'outschdr'. File 'outschdr' (Appendix 1.3) lists the tasks assigned to each processor and 'outstat' gives statistics on the allocation.

APPENDIX 1.1

{Input Computation Graph - ingraph}
This is the input file to program reduction.p. This graph is reduced based on the reduction criterion to obtain a large grain graph.

```
#1
1      {Node id. or code}
1.0    {Processing time}
       {Argument Nodes}
2      {Result Nodes}
3.0    {Local Communication Time}
6.0    {Bus Communication Time}

#2
2
5.0
1
3 4 5
2.0 1.5 2.0
6.0 4.0 3.0

#3
3
5.0
2
6 7 8 9
2.0 2.0 2.0 3.0
4.0 4.0 3.0 3.7

#4
4
8.0
2
21
3.0
3.8

#5
5
1.0
2
21 27
1.0 1.0
3.0 2.0

#6
6
2.0
3
10
1.2
2.2

#7
7
2.0
3
10
1.7
3.7

#8
8
2.0
3
11
0.8
2.8

#9
9
4.0
3
29
1.0
2.0

#10
10
1.0
6 7
12
1.0
2.0

#11
11
1.0
8
12
1.0
2.0

#12
12
5.0
10 11
13 14
1.0 1.0
2.0 1.8

#13
13
2.0
12
15 16
2.0 1.0
5.0 2.0

#14
14
2.0
12
18 19
0.1 2.2
1.1 4.2

#15
15
2.0
13
17
0.9
1.9
```

```
#16
16
2.0
13
17
1.0
2.0

#17
17
4.0
15 16
29
3.0
6.0

#18
18
2.0
14
20
1.0
4.0

#19
19
2.0
14
20
1.0
2.0

#20
20
4.0
18 19
29
1.0
2.4

#21
21
1.0
4 5
22 25
1.0 3.0
2.0 4.0

#22
22
3.0
21
23
2.0
3.0

#23
23
4.0
22
24
1.0
3.0

#24
24
2.0
23
27
2.0
3.0

#25
25
5.0
21
26
1.0
3.0

#26
26
6.0
25
27
3.0
4.0

#27
27
1.0
5 24 26
28
1.0
3.0

#28
28
4.0
27
29
2.0
3.0

#29
29
10.0
17 20 9 28
30
2.0
5.0

#30
30
1.0
29
```

APPENDIX 1.2

{Reduced graph - outgraph4}
This is the output of reduction.p
after the graph is parameterized
and then reduced. This graph is
the input to the allocation
program allocation.p.

```
NO OF NODES          23
node             1
funct                  2,1
proctime                      20.000
narg                           0
nres                           3
{results}     2        3        4
{loc comm}  0.500    0.500    0.500
{bus comm}   5.000    5.000    5.000

node             2
funct                  3
proctime                      10.000
narg                           1
{arg node}      1
nres                           4
     5          6          9          7
 0.500      0.500      0.500      0.500
 5.000      5.000      5.000      5.000

node             3
funct                  4
proctime                      10.000
narg                           1
          1
nres                           1
         19
      0.500
      5.000

node             4
funct                  5
proctime                      10.000
narg                           1
          1
nres                           2
         19         22
      0.500      0.500
      5.000      5.000

node             5
funct                  6
proctime                      10.000
narg                           1
          2
nres                           1
          8
      0.500
      5.000

node             6
funct                  7
proctime                      10.000
narg                           1
          2
nres                           1
          8
      0.500
      5.000

node             7
funct                  9
proctime                      10.000
narg                           1
          2
nres                           1
         23
      0.500
      5.000

node             8
funct                  10
proctime                      10.000
narg                           2
          5          6
nres                           1
         10
      0.500
      5.000

node             9
funct                  11,8
proctime                      20.000
narg                           1
          2
nres                           1
         10
      0.500
      5.000

node            10
funct                  12
proctime                      10.000
narg                           2
          8          9
nres                           2
         11         12
      0.500      0.500
      5.000      5.000

node            11
funct                  13
proctime                      10.000
narg                           1
         10
nres                           2
         13         14
      0.500      0.500
      5.000      5.000

node            12
funct                  14
proctime                      10.000
narg                           1
         10
nres                           2
         16         17
      0.500      0.500
      5.000      5.000
```

```
node            13
funct                     15
proctime                          10.000
narg                              1
          11
nres                              1
          15
      0.500
      5.000

node            14
funct                     16
proctime                          10.000
narg                              1
          11
nres                              1
          15
      0.500
      5.000

node            15
funct                     17
proctime                          10.000
narg                              2
          13             14
nres                              1
          23
      0.500
      5.000

node            16
funct                     18
proctime                          10.000
narg                              1
          12
nres                              1
          18
      0.500
      5.000

node            17
funct                     19
proctime                          10.000
narg                              1
          12
nres                              1
          18
      0.500
      5.000

node            18
funct                     20
proctime                          10.000
narg                              2
          16             17
nres                              1
          23
      0.500
      5.000

node            19
funct                     21
proctime                          10.000
narg                              2
          3              4
nres                              2
         20             21
      0.500          0.500
      5.000          5.000

node            20
funct                     24,23,22
proctime                          30.000
narg                              1
          19
nres                              1
          22
      0.500
      5.000

node            21
funct                     26,25
proctime                          20.000
narg                              1
          19
nres                              1
          22
      0.500
      5.000

node            22
funct                     28,27
proctime                          20.000
narg                              3
          4              20          21
nres                              1
          23
      0.500
      5.000

node            23
funct                     30,29
proctime                          20.000
narg                              4
          15             18          7            22
nres                              0
```

APPENDIX 1.3

```
{Allocation of tasks to processors - outschdr}
This output file from allocation.p indicates
which tasks are allocated to which processors.
It also gives the starting time and finishing
time for the tasks when executed in the
reverse schedule.
                            Task #     Start       Finish Time

PROCESSOR          1:        23(        0.0,        20.0)
                             18(       20.0,        30.5)
                             17(       30.5,        41.0)
                             16(       41.0,        51.5)
                             12(       51.5,        62.5)
                             22(       62.5,        83.0)
                             20(       83.0,       113.5)
                              9(      113.5,       138.5)
                              7(      138.5,       149.0)
                              2(      149.0,       170.0)
No of tasks            10

PROCESSOR          2:        15(       20.0,        35.0)
                             14(       35.0,        45.5)
                             13(       45.5,        56.0)
                             11(       56.0,        67.0)
                             10(       67.0,        82.5)
                              8(       82.5,        93.0)
                             21(       93.0,       118.0)
                              6(      118.0,       128.5)
                              5(      128.5,       139.0)
                             19(      139.0,       154.5)
                              4(      154.5,       170.0)
                              3(      170.0,       180.5)
                              1(      180.5,       206.5)
No of tasks            13
```

APPENDIX 2

```
{**************************************************************************
       REDUCTION                  8/18/1985              T.M.RAVI
                        (c) by T. M. Ravi
                               1985
 ************************************************************************** }
program reduction(input, output);
{
        This program reads in a program graph and reduces it based on
        communication and processing time criterion alone. We assume
        that the input graph is a single input-single output graph.

        INPUT:
               files   ingraph -  Program graph given by user
        OUTPUT:
               files   outgraph1 - Original graph without reduction
                       outgraph2 - Original graph with parameterized timing
                                   if parameterization option been excercised
                       outgraph3 - Graph after upward reduction
                       outgraph3 - Graph after downward reduction
                                   Final reduced graph
        PROCEDURE:
               upreduc         - Reduces the graph starting at the result node
               dnreduc         - Reduces the graph starting at the entry node
               datain          - Inputs the graph from file ingraph
               dataout         - Prints the current graph
               remnodes        - Removes nodes from tree  structure which are
                                 no longer present
               parameterize    - Allows parameterization of processing time,
                                 local and bus communication time.
}
const
    maxnodes = 130;             {maximum number of nodes in program graph}
    maxfunchar = 100;           {maximum characters in definition of function}


type
    tmaxnodes= 0..maxnodes;
    tfunct= packed array[1..maxfunchar] of char;
    targ=^link1;
    link1= record
            no:tmaxnodes;              {index of argument node      }
            dir:char;                  {arg label,f-forward arc,b-backward arc}
            next:targ;                 {pointer to next arg        }
        end;
    tres=^link2;
    link2= record
            no:tmaxnodes;              {identifier of the node                }
            dir:char;                  {res label,f-forward arc,b-backward arc}
            commtime:real;             {communication time of result arc     }
            bustime:real;              {bus communication time of result arc  }
            next:tres;                 {pointer to next res         }
        end;
    tnode = record                      {structure for representation of      }
                                        {each node belonging to the graph     }
              funct: tfunct;            {description of node                  }
              narg: integer;            {number of arguments                  }
              arg: targ;                {pointer to arguments                 }
              nres: integer;            {number of results                    }
              res: tres;                {pointer to results                   }
              proctime: real;           {processing time                      }
            end;
    typetree = array [1..maxnodes] of tnode; {tree = collection of nodes      }
```

```
var

    tree: typetree;                     {array to store program graph    }
    nonodes: tmaxnodes;                 {total number of nodes initially }
    newnonodes: tmaxnodes;              {total number of nodes   }
    entrynode: tmaxnodes;               {index of entry node}
    out:text;                           {var for text files }


{*************************************************************************
                                                                 DATAIN
 ************************************************************************* }
procedure datain(var tree:typetree;var nonodes:tmaxnodes);
{

         Procedure to input the program graph from file ingraph.
         Ingraph has the nodes listed in order. An example of a node:

                 #2                     (delimiter between nodes)
                 2OR                    (node index 2 with function OR)
                 1.0                    (Processing time)
                 1 30b                (arg. 1 and 30 with backward arc from 30)
                 3 4                    (Result nodes)
                 1.5 1.8                (Local communication times for results)
                 4.1 4.4                (Bus comm. times)

                 #3                     (next node ....)


         INPUT:  file ingraph

         OUTPUT:
                 tree     -  tree (graph) as an array of nodes.
                 nonodes  -  no. of nodes in initial graph.
}
var     i,j,l: integer;
        p:real;
        inp:text;
        tmpchar:char;
        firstptr,ptr,prevptr:targ;
        firstqtr,qtr,prevqtr:tres;
begin
  reset (inp,'ingraph');
  nonodes:=0;
  while not eof(inp) do begin
    read(inp,tmpchar);
    if tmpchar<> '#' then
      writeln('ERROR 1 in DATAIN - New node description should start with #')
    else begin
      nonodes:=nonodes+1;
      readln(inp,j);   {index of new node}
      with tree[j] do begin
                          {funct[1] & funct[2] are reserved. The function
                           starts from funct[3]}
        funct[1]:='U';i:=3;  {funct[1] can be 'X', 'D' or 'U'}
                          {'X' indicates that the node no longer exists &
                           'D' & 'U' are for book-keeping purposes}
        while not eoln(inp) do begin
          read(inp,tmpchar);
          funct[i]:=tmpchar;  {read the function and place it starting funct[3]}
          i:=i+1;
        end;
        funct[i]:=' ';
```

ALLOCATION FOR THE SANDAC MULTIPROCESSOR SYSTEM

T. M. Ravi
M. D. Ercegovac

February 1986
CSD-860059

```
 {If 1st char. of function is 'S' then the function is SWITCH, if it is
  'O' then the function is 'OR' else the function is neither ('N').
  funct[2] is used to indicate whether a function is a SW,OR or neither}

if funct[3]='O' then funct[2]:='O' else
  if funct[3]='S' then funct[2]:='S' else
  funct[2]:='N';
readln(inp,proctime);                         {processing time of node}

{read arguments of this node. The arguments are stored in a linked list}
l:=0;i:=1; {l counts the no. of aruments}
while ((not eoln(inp)) and (i<>0)) do begin
  read(inp,i);
  if i <> 0 then begin
    l:=l+1;
    new(ptr);
    if l=1 then firstptr:=ptr else prevptr^.next:=ptr;
    ptr^.no:=i;
    ptr^.next:=nil;
    prevptr:=ptr;
    if not eoln(inp) then begin
      read(inp,tmpchar);
      {if the arg. is a backward arc, i.e., coming from below this node
       (possible only for an OR node) then the input should indicate it
        example  30b indicates that the argument node is no. 30 and the
        arc from 30 to this node is a backward arc}

      if tmpchar='b' then ptr^.dir:='b' else
      if tmpchar<>' ' then writeln('ERROR 2 in DATAIN')
         else ptr^.dir:='f';  {direction is forward if not backward}
    end;
  end;
end;
readln(inp);
if l<>0 then arg:=firstptr;
narg:=l;

{read in the result nodes}
l:=0;i:=1;
while ((not eoln(inp)) and (i<>0)) do begin
  read(inp,i);
  if i <> 0 then begin
    l:=l+1;
    new(qtr);
    if l=1 then firstqtr:=qtr else prevqtr^.next:=qtr;
    qtr^.no:=i;
    qtr^.next:=nil;
    prevqtr:=qtr;
    if not eoln(inp) then begin
      read(inp,tmpchar);
      if tmpchar='b' then qtr^.dir:='b' else
      if tmpchar<>' ' then writeln('ERROR 3 in DATAIN')
         else qtr^.dir:='f';
    end;
  end;
end;
readln(inp);
if l<>0 then res:=firstqtr;
nres:=l; {set nres=the counter l}

{local and bus communication time are read from input graph. They will
 not be used if the parameterize option is chosen by the user }

{read in the local communication time for each result. Note that for
```

```
         each result the input should have a corresponding local
         communication time. }
        qtr:=res;
        if nres > 0 then begin
          for l:=1 to nres do begin
            read(inp,p);
            qtr^.commtime:=p;
            qtr:=qtr^.next;
          end;
          readln(inp);
        end;
        {read in the bus communication time for each result}
        qtr:=res;
        if nres > 0 then begin
          for l:=1 to nres do begin
            read(inp,p);
            qtr^.bustime:=p;
            qtr:=qtr^.next;
          end;
          readln(inp);
        end;
        readln(inp);
      end;
    end;
  end;
end;    {datain}

{****************************************************************************
                                                                   REMNODES
 **************************************************************************** }
procedure remnodes( var tree:typetree;var nonodes:tmaxnodes);
{

         Procedure to remove nodes which no longer exist (i.e., that have been
         combined). Basically to clean up the tree data structure.

         INPUT:
                nonodes         -  number of nodes including notes which are no
                                   longer valid
                tree            -  tree data structure with valid and
                                   invalid nodes
         OUTPUT:
                nonodes         -  actual numer of valid nodes
                tree            -  tree structure with only valid nodes
}
var     i,j,k: integer;
        actnonodes:tmaxnodes;    {actual number of nodes}
        ptr:targ;
        qtr:tres;
        labmap: array[tmaxnodes] of tmaxnodes;  {array to map old node index and

begin
  j:=0;
  for i:=1 to nonodes do
    if tree[i].funct[1] <> 'X' then begin
       {nodes with funct[1] = 'X' are no longer valid nodes}
      j:=j+1;
      labmap[i]:=j;
  end;
  actnonodes:=j;
  j:=0;
  for i:=1 to nonodes do begin
    with tree[i] do begin
      if funct[1]<>'X' then begin
```

```
        j:=j+1;
        tree[j].funct:=funct;
        tree[j].proctime:=proctime;
        tree[j].narg:=narg;
        tree[j].arg:=arg;
        ptr:=arg;
        if narg>0 then
          for k:=1 to narg do begin
            ptr^.no:=labmap[ptr^.no];
            ptr:=ptr^.next;
        end;
        tree[j].nres:=nres;
        tree[j].res:=res;
        qtr:=res;
        if nres>0 then
          for k:=1 to nres do begin
            qtr^.no:=labmap[qtr^.no];
            qtr:=qtr^.next;
        end;
      end;
    end;
  end;
  entrynode:=labmap[entrynode];
  nonodes:=actnonodes;
end; {remnodes}

{***************************************************************************
                                                                   DATAOUT
 *************************************************************************** }
procedure dataout(tree:typetree;nonodes:tmaxnodes;newnonodes:tmaxnodes);
{

         Procedure to output the program graph to  a file set to text var out.

         INPUT:
                nonodes         - total no. of nodes in the graph
                tree            - graph with nodes in an array
         OUTPUT:
                out             - output in file eq. to variable out
}
var     i,j: integer;
        ptr:targ;
        qtr:tres;

begin
  writeln(out,'NO OF NODES',newnonodes);
  for i:=1 to nonodes do
    if tree[i].funct[1] <> 'X' then {if node is valid}
    with tree[i] do begin
      writeln(out,'node ',i);
      funct[1]:=' ';funct[2]:=' ';
      writeln(out,'funct              ',funct);
      writeln(out,'proctime               ',proctime:10:3);
      writeln(out,'narg               ',narg);
      ptr:=arg;
      if narg > 0 then begin
        for j:=1 to narg do begin
          write(out,ptr^.no) ;
          if ptr^.dir='b' then write(out,'b');
          ptr:=ptr^.next;
        end;
        writeln(out);
      end;
      writeln(out,'nres               ',nres);
      qtr:=res;
      if nres > 0 then begin
        for j:=1 to nres do begin
          write(out,qtr^.no) ;
          if qtr^.dir='b' then write(out,'b');
          qtr:=qtr^.next;
        end;
        writeln(out);
      end;
      qtr:=res;
      if nres > 0 then begin
        for j:=1 to nres do begin
          write(out,qtr^.commtime:10:3) ;
          qtr:=qtr^.next;
        end;
        writeln(out);
      end;
      qtr:=res;
      if nres > 0 then begin
        for j:=1 to nres do begin
          write(out,qtr^.bustime:10:3) ;
          qtr:=qtr^.next;
        end;
        writeln(out);
      end;
      writeln(out);
    end;
end;    {dataout}


{****************************************************************************
                                                       PARAMETERIZE
 **************************************************************************** }
procedure parameterize(var tree:typetree;nonodes:tmaxnodes);
{

         Procedure to parameterize the processing time and communication times
         in the program graph. Procedure asks if parameterization is required
         and if so requests for the parameters. If parameterization option is
         used then the times in the graph are overruled. If however we only
         want to parameterize the communication times then if we assign a
         negative parameter to the processing time then the processing times for
         the nodes will be taken from the input graph data


}
var      i,j: integer;
         tmpchar:char;
         pr,buscomm,loccomm:real;
         qtr:tres;


begin
  write ('Want to parameterize Communication  and Processing times (y or n) ?');
  readln(tmpchar);
  if ((tmpchar = 'y') or (tmpchar='Y')) then begin
    write ('Processing time:'); readln(pr);
    write ('Local Communication time:'); readln(loccomm);
    write ('Bus Communication time:'); readln(buscomm);
    writeln('Pr.time = ',pr);
    writeln('LocComm.time = ',loccomm);{All arcs are given this local
                                                                  comm. time}
    writeln('BusComm.time = ',buscomm);{All arcs are given this bus comm. time }
    for i:=1 to nonodes do begin
```

```
      with tree[i] do begin
        if pr >= 0 then proctime:=pr; {All nodes are given this proc. time if it
                                   is positive else retain original proc. times}
        if nres>0 then begin
          qtr:=res;
          for j:=1 to nres do begin
            qtr^.commtime:=loccomm;
            qtr^.bustime:=buscomm;
            qtr:=qtr^.next;
          end;
        end;
      end;
    end;
   end;
end;{parameterize}


{***************************************************************************
                                                                     UPREDUC
 *************************************************************************** }
procedure upreduc(var tree:typetree; index:tmaxnodes; var newnonodes:tmaxnodes);
{
         Starting from node index this recursive procedure checks if the
         condition for combining the argument nodes and this node is
         satisfied. If it is then the functions of the argument nodes
         are copied to the index node. The index node's arguments will
         now be the arguments of the arguments. The result field of the
         arguments of the arguments has to be modified to reflect new
         results. If due to reduction we encounter two arcs between a
         pair of nodes we sum the comm times and replace them by a single
         arc. Note no upward reduction of OR nodes.

         INPUT:
                index           - present node which is being analyzed
                tree            - graph
         OUTPUT:
                tree            - graph after upward reduction
         PROCEDURE:
                upreduc         - recursive
}
var     n,i,k,m,l: integer;
        cond: real;
        singres: boolean;
        maxtime,sumproctime,largres: real;
        tnarg: integer;
        prevptr,ptr,rtr,firstptr,tptr:targ;
        prevqtr,qtr:tres;


begin
  ptr:=tree[index].arg;
  with tree[index] do begin
    if ((narg>0) and (funct[1]<>'D')) then
    if funct[1] = 'X' then
    writeln ('ERROR in UPREDUC - reference to invalid (nonexistent) node') else
    if funct[2] = 'O' then      {no upward reduction of OR nodes}
      for i:=1 to narg do begin
        if ptr^.dir <> 'b' then upreduc(tree,ptr^.no,newnonodes);{Only OR nodes
                                            can have backward arcs as argument}
        ptr:=ptr^.next;
      end
    else begin
         maxtime:=0; sumproctime:=0; ptr:=arg; singres:=true;
```

```
i:=1;k:=narg;
while ((ptr<>nil) and (singres=true)) do begin
  if tree[ptr^.no].nres >1 then begin
    qtr:=tree[ptr^.no].res;
    n:=0; largres:=0;
    m:=tree[ptr^.no].nres;
    for l:=1 to m do begin
      if qtr^.no = index then begin
        n:=n+1;
        if qtr^.commtime>largres then largres:=qtr^.commtime;
      end else singres:=false;
      qtr:=qtr^.next;
    end;
    if n=m then begin
      tree[ptr^.no].res^.commtime:=largres;
      tree[ptr^.no].res^.next:=nil;
      tree[ptr^.no].nres:=1;
      tptr:=ptr;tptr:=tptr^.next;prevptr:=ptr;
      while tptr<> nil do begin
        if tptr^.no = ptr^.no then begin
          prevptr^.next:=tptr^.next;
          narg:=narg-1;
        end else prevptr:=tptr;
        tptr:=tptr^.next;
      end;
    end;
  end;
  if singres=true then begin
    sumproctime:=sumproctime+tree[ptr^.no].proctime;
    if ( tree[ptr^.no].proctime +tree[ptr^.no].res^.commtime)> maxtime
    then
      maxtime:=tree[ptr^.no].proctime + tree[ptr^.no].res^.commtime;
  end;
  i:=i+1;
  ptr:=ptr^.next;
end;
ptr:=arg;
cond:=maxtime-sumproctime;        {compresion condition}
 {combination of node and its arguments}
if ((cond<=0) or (singres=false)) then         {no compresion}
  for i:=1 to narg do begin
    upreduc(tree,ptr^.no,newnonodes);
    ptr:=ptr^.next;
  end
else begin                                     {compresion}
  tnarg:=0;tptr:=arg;firstptr:=nil;
  m:=0;
  repeat m:=m+1 until funct[m]=' ';
  for i:=1 to narg do begin
    if tree[tptr^.no].narg > 0 then  begin    {new arg for index}
      tnarg:=tnarg+tree[tptr^.no].narg;
      if firstptr=nil then begin
        rtr:=tree[tptr^.no].arg;
        firstptr:=rtr;
      end else begin
        rtr^.next:=tree[tptr^.no].arg;
        rtr:=rtr^.next;
      end;
      for k:=1 to tree[tptr^.no].narg do begin  {res of arg of args
                                                          modified}
        qtr:=tree[rtr^.no].res;
          for l:=1 to tree[rtr^.no].nres do begin
            if qtr^.no=tptr^.no then qtr^.no:=index;
            qtr:=qtr^.next;
```

```
                   end;
                 if rtr^.next <> nil then rtr:=rtr^.next;
               end;
             end;
             k:=2;                      {copy functions of arg to index node}
             funct[m]:=',';
             repeat m:=m+1; k:=k+1; funct[m]:=tree[tptr^.no].funct[k]
             until tree[tptr^.no].funct[k]=' ';
             if tree[tptr^.no].funct[2]='O' then funct[2]:='O';
             if tptr^.no=entrynode then entrynode:=index;
             tree[tptr^.no].funct[1]:='X';     {Arg node no longer part of tree}
             newnonodes:=newnonodes-1;
             tree[tptr^.no].arg:=nil;
             tptr:=tptr^.next;
           end;
           arg:=firstptr;
           narg:=tnarg;                     {No. of arg is sum of narg of args}
           proctime:=proctime+sumproctime;  {new proc. time is sum of proc.
                                               times of all the nodes combined}
           upreduc(tree,index,newnonodes);  {Try reduction with new arguments }
         end;
     end;
     if funct[1] <> 'X' then funct[1]:='D';          {Mark it as observed }
      {'D' indicates that upreduc has encountered this node}
   end;
end;    {upreduc}

{***************************************************************************
                                                                  DNREDUC
 *************************************************************************** }
procedure dnreduc(var tree:typetree; index: tmaxnodes;var newnonodes:tmaxnodes);
{

         Starting from node index this recursive procedure checks if the
         condition for combining the result nodes and this node is
         satisfied. If it is then the functions of the result nodes
         are copied to the index node. The index node's results will
         now be the results of the result. The result field of the
         result of the result has to be modified to reflect new
         results. If due to reduction we encounter two arcs between a
         pair of nodes we sum the comm. times and replace them by a single
         arc.

         INPUT:
                index          -
                tree           -
         OUTPUT:
                tree           -
         PROCEDURE:
                dnreduc        - recursive
}
var     n,i,k,m,l: integer;
        cond: real;  {compression condition - compress if >0 }
        singarg: boolean;
        maxtime,sumproctime,largarg: real;
        tnres: integer;
        ptr,prevptr:targ;
        qtr,rtr,firstqtr,tqtr,prevqtr:tres;


begin
  qtr:=tree[index].res;
  with tree[index] do begin
    if ((nres>0) and (funct[1]<>'U')) then
if funct[1] = 'X' then
writeln ('ERROR  in DNREDUC - invalid node encountered') else
if ((funct[2] = 'S') or (funct[3] = 'S')) then{no down reduction for SWITCH}
  for i:=1 to nres do begin
    if qtr^.dir <> 'b' then dnreduc(tree,qtr^.no,newnonodes);
    qtr:=qtr^.next;
  end
else begin
      maxtime:=0; sumproctime:=0; qtr:=res; singarg:=true;
      i:=1;k:=nres;

       {singarg will be true if the result nodes of index node have
        only one argument which is the index node or all its arguments
        are the index node. Even though we don't admit two arcs in the same
        direction between the same pair of nodes initially, this can occur
        after combinations}
      while ((qtr<>nil) and (singarg=true)) do begin
        if tree[qtr^.no].narg >1 then begin
           {if the result has more than one argument}
          ptr:=tree[qtr^.no].arg;
          n:=0;largarg:=0;
          m:=tree[qtr^.no].narg;
          for l:=1 to m do begin
            if ptr^.no= index then begin
              n:=n+1;
            end else singarg:=false;
            ptr:=ptr^.next;
          end;
          if n=m then begin
            {if all the arguments of the result node are the index node,
             i.e., the node under consideration}
            {if parallel arcs from index to result then replace by a single
             arc}

            trée[qtr^.no].narg:=1;
            tree[qtr^.no].arg^.next:=nil;
            tqtr:=qtr;tqtr:=tqtr^.next;prevqtr:=qtr;
            largarg:=0;
            while tqtr<>nil do begin
              if tqtr^.no=qtr^.no then begin
                prevqtr^.next:=tqtr^.next;
                nres:=nres-1;
                if tqtr^.commtime>largarg then largarg:=tqtr^.commtime;
              end else prevqtr:=tqtr;
              tqtr:=tqtr^.next;
            end;
            qtr^.commtime:=largarg;
          end;
        end;
        if singarg=true then begin
          sumproctime:=sumproctime+tree[qtr^.no].proctime;
          if ( tree[qtr^.no].proctime +qtr^.commtime)> maxtime then
            maxtime:=tree[qtr^.no].proctime + qtr^.commtime;
        end;
        i:=i+1;
        qtr:=qtr^.next;
      end;
      qtr:=res;
      cond:=maxtime-sumproctime;        {compresion condition}
      if ((cond<=0) or (singarg=false)) then          {no compresion}
        for i:=1 to nres do begin
          dnreduc(tree,qtr^.no,newnonodes);
          qtr:=qtr^.next;
        end
```

```
          else begin                  {compresion}
            {combination of node and its results}
            tnres:=0;tqtr:=res;firstqtr:=nil;
            m:=0;
            repeat m:=m+1 until funct[m]=' ';
            for i:=1 to nres do begin
              if tree[tqtr^.no].nres > 0 then  begin
                tnres:=tnres+tree[tqtr^.no].nres; {no. of results of results}
                if firstqtr=nil then begin
                  rtr:=tree[tqtr^.no].res;  {result of the result}
                  firstqtr:=rtr;  {firstqtr will be the new result}
                end else begin
                  rtr^.next:=tree[tqtr^.no].res;
                  rtr:=rtr^.next;
                end;
                for k:=1 to tree[tqtr^.no].nres do begin
                  ptr:=tree[rtr^.no].arg;
                    for l:=1 to tree[rtr^.no].narg do begin
                      if ptr^.no=tqtr^.no then ptr^.no:=index;
                      ptr:=ptr^.next;
                    end;
                  if rtr^.next <> nil then rtr:=rtr^.next;
                end;
              end;
            k:=2;
            funct[m]:=',';
            {copy the functions}
            repeat m:=m+1; k:=k+1; funct[m]:=tree[tqtr^.no].funct[k]
            until tree[tqtr^.no].funct[k]=' ';
            if ((tree[tqtr^.no].funct[2]='S') or
                (tree[tqtr^.no].funct[3]='S')) then funct[2]:='S';
            tree[tqtr^.no].funct[1]:='X';
            newnonodes:=newnonodes-1;
            tree[tqtr^.no].res:=nil;
            tqtr:=tqtr^.next;
          end;
          res:=firstqtr;
          nres:=tnres;
          proctime:=proctime+sumproctime;
          dnreduc(tree,index,newnonodes);
        end;
      end;
      if funct[1] <> 'X' then funct[1]:='U';
       {Node with funct[1] = 'U' indicates that upreduc has seen this
         node already}
    end;
end;    {dnreduc}


{***************************************************************************
                 m a i n    p r o g r a m
 *************************************************************************** }
begin
  nonodes:=0;entrynode:=1;   {entry node is the single entry node of the graph}
  datain(tree,nonodes);      {read the input graph from "ingraph"}
  newnonodes:=nonodes;
  rewrite(out,'outgraph1');
  dataout(tree,nonodes,newnonodes); {write graph to text var out}
  parameterize (tree,nonodes); {option to give general time parameters}
  rewrite(out,'outgraph2');
  dataout(tree,nonodes,newnonodes);
  upreduc(tree,nonodes,newnonodes);{upward reduction of nodes starting from
  rewrite(out,'outgraph3');
  dataout(tree,nonodes,newnonodes);
  dnreduc(tree,entrynode,newnonodes);{downward reduction of nodes starting from
                                         entry node}
  remnodes(tree,nonodes);          {remove nodes which are no longer in the graph}
  rewrite(out,'outgraph4');
  dataout(tree,nonodes,newnonodes);
end.
```

APPENDIX 3

```
{***********************************************************************
       ALLOCATION                    9/4/1985               T.M.RAVI

                         (c) Copyright by T. M. Ravi
                                   1985
 *********************************************************************** }
program allocation(input, output);
{
         This program reads in a program graph and reduces it based on
         communication and processing time criterion alone. We assume
         that the input graph is a single input-single output graph.

         INPUT:
                files   outgraph2 - Original graph without reduction
                  or    outgraph4 - Graph after reduction
                                    Both files outgraph2 & outgraph4 are output
                                    files from program reduction.p

         OUTPUT:
                files   outgraph5 - Graph selected to be allocated
                        outgraph6 - Graph which is reverse of outgraph6
                        outgraph7 - Graph of outgraph6 with critical path of
                                    nodes  indicated
                        outgraph8 - Reverse Graph indicating which processor
                                    each node has been allocated to
                        outlist   - List of tasks yet to be allocated ordered
                                    in decreasing order of critical path
                        outschdr  - Tasks assigned to each processor
                        outstat   - Statistics on this allocation


          PROCEDURE:
                 revgraph         - Reverse graph by changing direction of arcs
                 graphin          - Read the graph to be allocated from outgraph2
                                    or outgraph4
                 revgraph         - Reverse the graph, i.e., reverse the direction
                                    of arcs
                 dataout          - Prints the current graph
                 evalcp           - Evaluate the critical paths for all the nodes
                                    in the graph
                 setuplist        - Set up a list of nodes (tasks) ordered
                                    according to decreasing critical path
                 initproclist     - Initialize a list of processors
                 putproclist      - Place a processor which has been allocated a
                                    new task in the correct position in the proc.
                                    list
                 calcomtime       - Calculate the communication time (of
                                    arguments) for the task to be allocated
                 calseqextime     - Calculates execution time when we have only
                                    one processor
                 scheduler        - Main allocation algorithm
                 listout          - Print out the task list (list)
                 schdrout         - Print out the nodes (tasks) allocated to each
                                    processor along with start and finish time
                                    for the reverse schedule
                 stats            - Calculates and prints statistics for
                                    this particular allocation

 }
 const
     maxnodes = 100;              {maximum number of nodes in program graph}
     maxfunchar = 200;            {maximum characters in definition of function}
     maxnoproc  = 100;            {maximum number of processors}
```

```
type
   tmaxnodes= 0..maxnodes;
   tmaxnoproc= 0..maxnoproc;
   tfunct= packed array[1..maxfunchar] of char;
   tres=^link2;
   link2= record
           no:tmaxnodes;
           dir:char;                  {res label,f-forward arc,b-backward arc}
           commtime:real;             {local communication time of result arc}
           bustime:real;              {external communication time of result arc}
           next:tres;                 {pointer to next res        }
          end;
   tnode = record                        {structure for representation of     }
                                         {each node belonging to the graph    }
              funct: tfunct;             {description of node                 }
              narg: integer;             {number of arguments                 }
              arg: tres;                 {pointer to arguments                }
              nres: integer;             {number of results                   }
              res: tres;                 {pointer to results                  }
              proctime: real;            {processing time                     }
              sumbustime: real;          {sum of the bus communication times
                                                               of results     }

              criticalpath: real;        {processing time                     }
              procid: integer;           {processor to which node has been
                                                    allocated                 }

              tmax: real;                {time when this node completes
                                          execution                           }

            end;
    textime= record                      {time when task starts execution     }
                lower:real;              {time when task ebds execution       }
                upper:real;
              end;
    task=^link3;
    link3= record
             no:tmaxnodes;               {task no.}
             exectime:textime;           {details on the execution times of task}
             next:task;                  {next task}
             prev:task;                  {previous task}
         end;
    typelist= record
                top:task;  {top of task list}
                size:integer; {size of task list}
              end;
    tprocsch= record               {description of each processor}
                first:task;        {first task allocated to it}
                last:task;         {last task allocated to it}
                busytime:real;     {time to which it is busy}
                notasks:integer;{no. of tasks allocated to the processor}
              end;
    typeschdr= array [tmaxnoproc] of tprocsch;
    proclist =^link4;
    link4= record                             {index of processor}
             no:tmaxnoproc;                   {pointer to next processor}
             next:proclist;                   {pointer to previous processor}
             prev:proclist;
           end;
    typeidleproc= record
                    front:proclist;  {front of processor list}
                    back:proclist;   {back of processor list }
                  end;
    typetree = array [1..maxnodes] of tnode; {tree-collection of nodes}
  var
    idleproc: typeidleproc;    { ordered list of processors }
   schdr: typeschdr;           {description of each processor}
   list: typelist;           {list of tasks ordered by decreasing critical path }
   tree: typetree;                      {array to store program graph    }
   nonodes: tmaxnodes;                  {total number of nodes  }
   entrynode: tmaxnodes;                {index of entry node}
   exitnode: tmaxnodes;                 {index of exit node}
   noproc: tmaxnoproc;                  {total number of processors  }
   out:text;

{*************************************************************************
                                                                  GRAPHIN
 *********************************************************************** }
procedure graphin(var tree:typetree;var nonodes:tmaxnodes;var entrynode:tmaxnodes;var e
{

        Procedure to read the program graph from file outgraph2 or outgraph4.
        We have the option of allocating the original graph (outgraph2) or
        the preprocessed (reduced) graph (outgraph4).


        INPUT:
               Graph outgraph2 or outgraph4
        OUTPUT:
                tree      -  array to store the program graph
                nonodes   -  no. of nodes in the graph
                entrynode-   the top node in the graph. Node with no arguments
                exitnode -    bottom node in the graph. Node with no results

 }
 var     i,j,k,l: integer;
         inp:text;
         tmpchar:char;
         p:real;
         firstptr,ptr,prevptr:tres;
         firstqtr,qtr,prevqtr:tres;
 begin
   write('Input is original graph {o} or reduced graph {r} ');readln(tmpchar);
   if tmpchar='o' then begin
     writeln('Reading original graph');
     reset(inp,'outgraph2');
   end else begin
     writeln('Reading reduced graph');
     reset (inp,'outgraph4');
   end;
   for k:=1 to 11 do read(inp,tmpchar);
   readln(inp,nonodes);
   exitnode:=nonodes;
   for i:=1 to nonodes do begin
     for k:=1 to 4 do read(inp,tmpchar);
     readln(inp,j);
     if i=1 then entrynode:=i;
     with tree[i] do begin
       for k:=1 to 5 do read(inp,tmpchar);
       k:=1;
       while not eoln(inp) do begin
         read(inp,tmpchar);
         funct[k]:=tmpchar;
         k:=k+1;
       end;
       readln(inp);
       criticalpath:=-1; tmax:=0;  {these two will be calculated later}
       for k:=1 to 8 do read(inp,tmpchar);
       readln(inp,proctime);
       for k:=1 to 4 do read(inp,tmpchar);
```

```
      readln(inp,narg);
      if narg > 0 then begin
        for l:=1 to narg do begin
          new(ptr);
          if l=1 then firstptr:=ptr else prevptr^.next:=ptr;
          read(inp,ptr^.no);
          ptr^.next:=nil;
          prevptr:=ptr;
         { read(inp,tmpchar);
          if tmpchar='b' then ptr^.dir:='b' else
            if tmpchar='f' then ptr^.dir:='f' else
            if tmpchar<>' ' then writeln('ERROR 3') else ptr^.dir:='f';
         }
       end;
       readln(inp);
       arg:=firstptr;
      end;
      for k:=1 to 4 do read(inp,tmpchar);
      readln(inp,nres);
      if nres > 0 then begin
        for l:=1 to nres do begin
          new(qtr);
          if l=1 then firstqtr:=qtr else prevqtr^.next:=qtr;
          read(inp,qtr^.no);
          qtr^.next:=nil;
          prevqtr:=qtr;
         { read(inp,tmpchar);
          if tmpchar='b' then qtr^.dir:='b' else
            if tmpchar='f' then qtr^.dir:='f' else
            if tmpchar<>' ' then writeln('ERROR 3') else qtr^.dir:='f';
         }
       end;
       readln(inp);
       res:=firstqtr;
      end;
      if nres > 0 then begin
        qtr:=res;
        for j:=1 to nres do begin
          read(inp,qtr^.commtime) ;
          qtr:=qtr^.next;
        end;
        readln(inp);
      end;
      p:=0; sumbustime:=0;
      if nres > 0 then begin
        qtr:=res;
        for j:=1 to nres do begin
          read(inp,qtr^.bustime);
          p:=p+qtr^.bustime;
          qtr:=qtr^.next;
        end;
        readln(inp);
        sumbustime:=p;
      end;
      readln(inp);
    end;
  end;
end;    {graphin}

{***************************************************************************
                                                          REVGRAPH
 *************************************************************************** }
procedure revgraph(var tree:typetree;nonodes:tmaxnodes;var entrynode:tmaxnodes;
                    var exitnode:tmaxnodes);
{

        Procedure to reverse the program graph. The direction of the arcs
        is reversed. The arguments and results are interchanged. The entry
        node and exit node have to be interchanged.  Communication time
        is now associated with arguments and not results.


        INPUT:
                tree      - original graph to be allocated
                nonodes   - no. of nodes in original graph
                entrynode- top node in graph
                exitnode - bottom node in graph

        OUTPUT:
                tree      - graph with reversed arcs
                entrynode- old exitnode
                exitnode - old entrynode
}
var     i,j: integer;
        ptr:tres;
begin
  for i:=1 to nonodes do
    with tree[i] do begin
      j:=nres; ptr:=res;  {switch arguments and results}
      nres:=narg;res:=arg;
      narg:=j; arg:=ptr;
    end;
  entrynode:=nonodes; {switch entry and exit node}
  exitnode:=1;
end;{revgraph}

{****************************************************************************
                                                                  DATAOUT
 **************************************************************************** }
procedure dataout(tree:typetree;nonodes:tmaxnodes);
{

        Procedure to output the program graph to file set to text var out.

        INPUT:
                nonodes          - no of nodes i the graph
                tree             - program graph
        OUTPUT:
                out              - output in file eq. to variable out

}
var     i,j: integer;
        ptr:tres;
        qtr:tres;

begin
  writeln(out,'No Of Nodes',nonodes);
  for i:=1 to nonodes do
    if tree[i].funct[1] <> 'X' then
    with tree[i] do begin
      writeln(out,'node ',i);
      funct[1]:=' ';funct[2]:=' ';
      writeln(out,'funct                ',funct);
      writeln(out,'proctime                 ',proctime:10:3);
      writeln(out,'critical path            ',criticalpath:10:3);
      writeln(out,'tmax                 ',tmax:10:3);
      writeln(out,'processor #          ',procid);
      writeln(out,'narg                 ',narg);
      ptr:=arg;
      if narg > 0 then begin
        for j:=1 to narg do begin
          write(out,ptr^.no) ;
       {    if ptr^.dir='b' then write(out,'b') else if ptr^.dir='f' then
             write(out,'f') else writeln('ERROR in dataout');
       }
          ptr:=ptr^.next;
        end;
        writeln(out);
      end;
      ptr:=arg;
      if narg > 0 then begin
        for j:=1 to narg do begin
          write(out,ptr^.commtime:10:3) ;
          ptr:=ptr^.next;
        end;
        writeln(out);
      end;
      ptr:=arg;
      if narg > 0 then begin
        for j:=1 to narg do begin
          write(out,ptr^.bustime:10:3) ;
          ptr:=ptr^.next;
        end;
        writeln(out);
      end;
      writeln(out,'nres                ',nres);
      qtr:=res;
      if nres > 0 then begin
        for j:=1 to nres do begin
          write(out,qtr^.no) ;
       {    if ptr^.dir='b' then write(out,'b') else if ptr^.dir='f' then
             write(out,'f') else writeln('ERROR in dataout');
       }
          qtr:=qtr^.next;
        end;
        writeln(out);
      end;
      qtr:=res;
      if nres > 0 then begin
        for j:=1 to nres do begin
          write(out,qtr^.commtime:10:3) ;
          qtr:=qtr^.next;
        end;
        writeln(out);
      end;
      qtr:=res;
      if nres > 0 then begin
        for j:=1 to nres do begin
          write(out,qtr^.bustime:10:3) ;
          qtr:=qtr^.next;
        end;
        writeln(out);
      end;
      writeln(out);
    end;
end;    {dataout}

{***************************************************************************
                                                                    EVALCP
 *************************************************************************** }
procedure evalcp(var tree:typetree; index: tmaxnodes);
{
        This program evaluates the critical path of each node in the graph. The
        critical path of a node is equal to the maximum critical path of the
        result nodes + processing time of that node + the total communication
        time of all the results


        INPUT:
               index           - present node for which critical path evaluated
               tree            - graph  (in this case actually the reversed
                                 graph)
        OUTPUT:
               tree            - graph with critical paths
        PROCEDURE:
               evalcp          - recursive
}
var     i: integer;
        cond: boolean;
        maxpath: real;
        ptr:tres;
        qtr:tres;


begin
  with tree[index] do begin
    if nres=0 then begin
      criticalpath:=proctime+sumbustime; {bottom node has lowest critical path}
      ptr:=arg;
      for i:=1 to narg do begin
        evalcp(tree,ptr^.no);
        ptr:=ptr^.next;
      end;
    end else if nres=1 then begin
      criticalpath:=proctime+sumbustime+tree[res^.no].criticalpath;
      ptr:=arg;
      for i:=1 to narg do begin
        evalcp(tree,ptr^.no);
        ptr:=ptr^.next;
      end;
    end else begin
      qtr:=res; maxpath:=0; cond:=true;
      for i:=1 to nres do begin
        if tree[qtr^.no].criticalpath<>-1 then begin
          if maxpath<tree[qtr^.no].criticalpath then
            maxpath:=tree[qtr^.no].criticalpath;
        end else cond:=false;
        qtr:=qtr^.next;
      end;
      if cond=true then begin
        criticalpath:=proctime+sumbustime+maxpath;
        ptr:=arg;
        for i:=1 to narg do begin
          evalcp(tree,ptr^.no);
          ptr:=ptr^.next;
        end;
      end;
    end;
  end;
end; {evalcp}

{***************************************************************************
                                                            SETUPLIST
 *************************************************************************** }
```

```
procedure setuplist(var list:typelist;tree:typetree;nonodes: tmaxnodes);
{

        Build an ordered list of nodes in decreasing order of critical paths
        of nodes. In our case the nodes in the top of the reversed graph, i.e.,
        the nodes in the bottom of the original graph will be in the top of the
        list.

        INPUT:
               index          - present node for which critical path evaluated
               tree           - graph
        OUTPUT:
               list           - list in decreasing order of critical paths of
                                nodes

}
var     i,j: integer;
        found: boolean;
        endptr,tptr,temptr: task;

begin
  with list do begin
    tptr:=nil; size:=0;
    for i:= nonodes downto 1 do begin
      new(tptr);
      tptr^.no:=i;
      if i= nonodes then begin
        tptr^.next:=nil;
        top:=tptr;
        endptr:=tptr;
        tptr^.prev:=nil;
      end else begin
        j:=size; temptr:=endptr; found:=false;
        while ((j<>0) and (found<>true)) do begin
          if tree[i].criticalpath > tree[temptr^.no].criticalpath then begin
            temptr:=temptr^.prev;
            j:=j-1;
          end else found:=true;
        end;
        if temptr<>endptr then begin
          tptr^.next:=temptr^.next;
          tptr^.prev:=temptr;
          tptr^.next^.prev:=tptr;
          temptr^.next:=tptr;
        end else begin
          tptr^.prev:=temptr;
          temptr^.next:=tptr;
          endptr:=tptr;
          tptr^.next:=nil;
        end;
      end;
      size:=size+1; {total size of the list - i.e., no. of nodes in the list}
    end;
  end;
end; {setuplist}

{**************************************************************************
                                                              INITPROCLIST
 ************************************************************************** }
procedure initproclist(var idleproc:typeidleproc;var noproc: tmaxnoproc;
                                                    var schdr:typeschdr);

{       This procedure initializes the processor list. The processor list
        is a doubly linked list ordered according to which processor will
        next become free. scdr[i].busytime indicates till what time the
        processor i is busy. Front indicates the top of the list and back
        the bottom of the list. Initially as all processors are idle they
        can be in any random order.


        INPUT:
               index          -
               tree           -
        OUTPUT:
               idleproc       -  List of processors ordered in increasin order
                                 of their busy times

}
var     i: integer;
        prevptr,tptr: proclist;

begin
  write('No. of processors =');  {How many processors do we want to allocate
                                  the graph to ?}

  readln(noproc);
  if noproc>1 then
    for i:= noproc downto 1 do begin
      new(tptr);
      tptr^.no:=i;
      if i=noproc then begin
        idleproc.back:=tptr;     {idleproc.back is the bottom of the list and
                                  idleproc.back^.no is the processor with largest
                                  busytime}

      end else begin
        prevptr^.next:=tptr;
        tptr^.prev:=prevptr;
      end;
      prevptr:=tptr;
      if i=1 then idleproc.front:=tptr; {top of the processor list}
      schdr[i].busytime:=0; {initially processor i is idle}
      schdr[i].notasks:=0;  {initially processor i has no task assigned to it}
      schdr[i].first:=nil;
      schdr[i].last:=nil;
    end;
 end;{initproclist}


 {***************************************************************************
                                                              PUTPROCLIST
  *************************************************************************** }
 procedure putproclist(var idleproc:typeidleproc;var procptr: proclist;
                                                  noproc:tmaxnoproc);

  {
          Places the processor (to which a task has just been allocated)
          in the right place in the processor list. The processor list is a list
          of processors ordered according to decreasing busytime

          INPUT:
                 index          -
                 tree           -
          OUTPUT:
                 list           -

  }
  var     i: integer;
          tptr: proclist;

  begin
  tptr:=idleproc.back;  {start from bottom of the list}
  i:=noproc-1;
  while ((i > 0) and (schdr[tptr^.no].busytime >= schdr[procptr^.no].busytime))
    i:=i-1;
    tptr:=tptr^.next;
  end;
  if i=0 then begin  {if the proper place is the front of the list}
    idleproc.front^.next:=procptr;
    procptr^.prev:=idleproc.front;
    idleproc.front:=procptr;
  end else begin
    if tptr^.prev <> nil then tptr^.prev^.next:=procptr;
    procptr^.prev:=tptr^.prev;
    procptr^.next:=tptr;
    tptr^.prev:=procptr;
    if idleproc.back=tptr then idleproc.back:=procptr; {if the new position is
                                                         bottom of the list}
  end;
end; {putproclist}


{****************************************************************************
                                                        CALCOMTIME
 **************************************************************************** }
procedure calcomtime(tree:typetree;i:tmaxnodes;procno:tmaxnoproc;var newcomtime:real);
{
         This procedure calculates the communication time of a task depending
         on whether the arguments (for our reverse graph) have been allocated
         to the same processor (local) or to a different processor (bus).

         INPUT:
                index          -
                tree           -
         OUTPUT:
                list           -

}
var      j: integer;
         ptr:tres;

begin
  newcomtime:=tree[i].sumbustime;
  {assuming all are allocated to different processors}
  ptr:=tree[i].arg;
  for j:=1 to tree[i].narg do begin
    if tree[ptr^.no].procid = procno then {check if allocated to same processor}
      newcomtime:=newcomtime - ptr^.bustime + ptr^.commtime;
    ptr:=ptr^.next;
  end;
end;{calcomtime}


{****************************************************************************
                                                        CALSEQEXTIME
 **************************************************************************** }
procedure calseqextime(tree:typetree;nonodes:tmaxnodes);
{

         Procedure to  calculate the time for execution when we have only one
         processor.  The execution time will be the sum of processing time and
         local communication time of each node.


         INPUT:
```

```
        OUTPUT:
                tree     -  array to store the program graph
                nonodes  -  no. of nodes in the graph
}
var     i,j: integer;
        seqextime:real; {Execution time for a single processor}
        ptr:tres;
begin
  seqextime:=0;
  for i:=1 to nonodes do begin
    with tree[i] do begin
      seqextime:=seqextime+proctime;
      if narg > 0 then begin
        ptr:=arg;
        for j:=1 to narg do begin
          seqextime:=seqextime+ptr^.commtime;
          ptr:=ptr^.next;
        end;
      end;
    end;
   end;
   writeln('The response time for ',noproc,' processor is ',
                                  seqextime:10:3);
end;{calseqextime}


{*************************************************************************
                                                            SCHEDULER
 ************************************************************************* }
procedure scheduler(var idleproc:typeidleproc;var schdr: typeschdr;var tree:
                        typetree;nonodes:tmaxnodes;noproc:tmaxnoproc);

{
        This is the algorithm for allocation of tasks to processors. We select
        the first processor on the processor list and pick candidates based on
        critical path criterion from the task list which can be allocated to
        the processor. A task is selected (based on the criterion of saving
        communication time by allocating predecessors and successors to the
        same processor. This is repeated till all the tasks in the task list
        have been allocated.

        INPUT:
                index           -
                tree            -
        OUTPUT:
                list            -

        PROCEDURE:

                selcandidates   - Select the candidate tasks which can
                                  be allocated at the time specified
                choosetask      - Choose the task from the candidates which
                                  will result in maximum saving on allocation
                                  to the processor under consideration
 }
 const
        maxnocandidates = 40;  {Maximum no. of candidates allowed}
 type
        tmaxnocandidates = 0..maxnocandidates;
        tcandidate = array[tmaxnocandidates] of record
                                              saving:real;
                                              loc:task;
                                            end;

 var
```

```
        procptr,temptr: proclist;
        taskptr:task;
        newcomtime:real;
        candidate:tcandidate;   {candidates for allocation to a processor}
        nocandidates:tmaxnocandidates;  {no. of candidates}
        cpdeviation:real;             {deviation in critical path of processors}
        chosencandidate:tmaxnocandidates;   {th candidate chosen for allocation}


{************************************************************************
                                                               CHOOSETASK
 ************************************************************************ }
procedure choosetask(candidate:tcandidate;nocandidates:tmaxnocandidates;
                              procno:tmaxnoproc;var taskno:tmaxnocandidates);

{
        Choose the task from the candidates which on allocation to the
        processor under consideration will result in maximum savings in
        communication time.

        INPUT:
               index          -
               tree           -
        OUTPUT:
               taskno         -

}
var     i,j: integer;
        ptr:tres;
        max:real; {maximum saving}

begin
  for i:=1 to nocandidates do begin  {for each candidate calculate saving}
    ptr:=tree[candidate[i].loc^.no].arg;
    candidate[i].saving:=0;
    for j:=1 to tree[candidate[i].loc^.no].narg do begin
      if tree[ptr^.no].procid = procno then
        {if argument is allocated to the same processor then communication
         will be local}
        candidate[i].saving:=candidate[i].saving + ptr^.bustime - ptr^.commtime;
      ptr:=ptr^.next;
    end;
  end;
  max:=0;
   {find which candidate task results in maximum saving}
  for i:=1 to nocandidates do
    if candidate[i].saving > max then begin
      max:=candidate[i].saving;
      taskno:=i;
    end;
  if max<0.00001 then taskno:=1;
 end;{choosetask}
 {************************************************************************
                                                               SELCANDIDATES
  ************************************************************************ }
 procedure selcandidates(var candidate:tcandidate;
                         var nocandidates:tmaxnocandidates;list:typelist;
                         tree:typetree;procbusytime:real;cpdeviation:real);
 {

         This procedure selects candidates which can be allocated at time
         procbusytime from the list of tasks (list). The first criterion to be
         satisfied is that the task should be available for execution at
        procbusytime. For this we have to check if all the argument tasks have
        finished by procbusytime. The first task which satisfies this criterion
        in the task list is the first candidate. Any other tasks which satisfy
        the criterion and are within cpdeviation of the first candidates
        critical path is also chosen as a candidate.

        INPUT:
               index          -
               tree           -
        OUTPUT:
               list           -

}
var     i,j: integer;
        taskptr:task;
        ptr:tres;
        cond,firstfound:boolean;
        limit:real;

begin
    nocandidates:=0;
    i:= list.size; taskptr:=list.top;  {start at top of list}
    firstfound:=false;
     {attempting to find first candidate}
    while ((i > 0) and (firstfound <> true)) do begin
      if tree[taskptr^.no].narg > 0 then begin
        ptr:=tree[taskptr^.no].arg; cond:=true;
        for j:=1 to tree[taskptr^.no].narg do begin
          if ((tree[ptr^.no].tmax > procbusytime) or
                                              {arg. not completed by procbusytime}
              (tree[ptr^.no].tmax < 0.0001)) then
                                              {arg. not yet allocated}
            cond:=false;
          ptr:=ptr^.next;
        end;
        if cond = true then firstfound:=true;
      end else firstfound:=true;
      if firstfound = false then begin
        i:=i-1; taskptr:=taskptr^.next; {try next task on list}
      end else begin   {first candidate has been found}
        nocandidates:=1;
        candidate[nocandidates].loc:=taskptr;
      end;
    end;
    if firstfound=true then begin
      limit:=tree[taskptr^.no].criticalpath -
             cpdeviation;      {limit is the range of critical path
                                            where candidates are chosen}

      i:=i-1;
      taskptr:=taskptr^.next;              {now look at rest of tasks}
      while i > 0 do begin
        if tree[taskptr^.no].criticalpath >= limit then begin {within range ?}
          if tree[taskptr^.no].narg > 0 then begin
            ptr:=tree[taskptr^.no].arg; cond:=true;
            for j:=1 to tree[taskptr^.no].narg do begin
              if ((tree[ptr^.no].tmax > procbusytime) or {criterion satisfied ?}
                  (tree[ptr^.no].tmax < 0.0001)) then
                cond:=false;
              ptr:=ptr^.next;
            end;
            if cond = true then begin
              nocandidates:=nocandidates+1;
              candidate[nocandidates].loc:=taskptr;
            end;
```

```
}
var     i: integer;
        taskptr:task;

begin
  i:=list.size; taskptr:=list.top;
  while i > 0 do begin
    write(out,taskptr^.no);
    taskptr:=taskptr^.next;
    i:=i-1;
  end;
end; {listout}

{*************************************************************************
                                                                  SCHDROUT
 ************************************************************************* }
procedure schdrout(schdr:typeschdr;noproc:tmaxnoproc);
{
        Print the tasks assigned to each processor along with the starting
        time and finishing time of each task for the reverse schedule.


}
var     i,j: integer;
        taskptr:task;

begin
  if noproc >1 then
  for i:=1 to noproc do begin
    taskptr:=schdr[i].first;
    write(out,'PROCESSOR',i,':');
    for j:=1 to schdr[i].notasks do begin
      write(out,taskptr^.no);
      write(out,'(',taskptr^.exectime.lower:10:1,
                                  ',',taskptr^.exectime.upper:10:1,') ');

      taskptr:=taskptr^.next;
    end;
    writeln(out);
    writeln(out,'No of tasks ',schdr[i].notasks);
  end;
end; {schdrout}


{*************************************************************************
                                                                  STATS
 ************************************************************************* }
procedure stats(tree:typetree;nonodes:tmaxnodes;schdr:typeschdr;
                noproc:tmaxnoproc;exitnode:tmaxnodes);
{

         Procedure to collect statistics on the allocation


}
var     i,j: integer;
        ptr:tres;
        taskptr:task;
        totnarg,nbusarcs,nlocarcs:integer;
        totbustime,totproctime,totcommtime,totidletime:real;
begin
  writeln(out,'No Of Nodes',nonodes);
  totproctime:=0;totnarg:=0;nbusarcs:=0;
  nlocarcs:=0;totbustime:=0;totcommtime:=0;
  for i:=1 to nonodes do
    if tree[i].funct[1] <> 'X' then
```

```
        end else begin  {node in question has no arguments- hence no criterion
                                               to be satisfied}
          nocandidates:=nocandidates+1;
          candidate[nocandidates].loc:=taskptr;
        end;
      end;
      i:=i-1; taskptr:=taskptr^.next;
    end;
  end;
end; {selcandidates}


begin {scheduler}
  if noproc =1 then calseqextime(tree,nonodes) else begin
     {if only 1 processor then no need to schedule}
  {cpdeviation is the deviation in the critical path of the first candidate.
   All tasks within cpdeviation of the first candidate is also a candidate}

  write(' Deviation in critical path for selecting candidates =');
  readln(cpdeviation);
  writeln(' Deviation in critical path is =',cpdeviation:10:2);
  while list.size>0 do begin {while there are some unallocated tasks left}
    procptr:=idleproc.front; {at first we attempt to allocate a tasks to the
                              processor which becomes available first}
    nocandidates:=0;{select candidates which can be allocated to that processor}
    selcandidates(candidate,nocandidates,list,tree,schdr[procptr^.no].busytime,
                  cpdeviation);
    if nocandidates>0 then begin  {if we found some candidate tasks which can
                                   begin execution at the time when the first
                                   processor in the processor list becomes idle}

      {choose the task from the candidates which will result in the maximum
       saving of communication time on placing it in this processor}
      choosetask(candidate,nocandidates,procptr^.no,chosencandidate);
      taskptr:=candidate[chosencandidate].loc; {points to the chosen task}
      {exectime.lower indicates when that task starts execution
                                               in this reverse schedule}
      taskptr^.exectime.lower:=schdr[procptr^.no].busytime;
      {calculate the communication time that will be associated with the
      incoming arcs to the task being allocated}
      calcomtime(tree,taskptr^.no,procptr^.no,newcomtime);

       {updating the busy time of the processor}
      schdr[procptr^.no].busytime:= schdr[procptr^.no].busytime+
                tree[taskptr^.no].proctime+ newcomtime;
       {exectime.upper indicates when it will comlete execution}
      taskptr^.exectime.upper:=schdr[procptr^.no].busytime;

       {now remove the task which has been chosen for allocation from the list}
      list.size·=list.size-1;
      if taskptr <> list.top then begin
                         {if the chosen candidate is not on top of the list}
        taskptr^.prev^.next:= taskptr^.next;
        if taskptr^.next <> nil then taskptr^.next^.prev:=taskptr^.prev;
        taskptr^.next:=nil;
        taskptr^.prev:=nil;
      end else begin      {if chosen candidate is on top of the list}
        list.top:= taskptr^.next;
        if taskptr^.next <> nil then taskptr^.next^.prev:=nil;
        taskptr^.next:=nil;
        taskptr^.prev:=nil;
      end;

       {in the graph we mark which processor that node has been allocated to and
       at what time it will complete execution}
     tree[taskptr^.no].tmax:=schdr[procptr^.no].busytime;
     tree[taskptr^.no].procid:=procptr^.no;

     {update the schedule of the processor. we remove the task from the task
      list and add it to the schedule of that processor}
     if schdr[procptr^.no].notasks=0 then begin
       schdr[procptr^.no].first:=taskptr;
       schdr[procptr^.no].last:=taskptr;
     end else begin
       schdr[procptr^.no].last^.next:=taskptr;
       taskptr^.prev:=schdr[procptr^.no].last^.prev;
       schdr[procptr^.no].last:=taskptr;
     end;
     {increase the no. of tasks in that processor}
     schdr[procptr^.no].notasks := schdr[procptr^.no].notasks +1;

     {after having chosen a task to be allocated to the first processor in the
      processor list we remove it from the front of the list}
     idleproc.front:=idleproc.front^.prev;
     procptr^.next:=nil;
     procptr^.prev:=nil;
     {place processor at correct position according to busytime in proc. list}
     putproclist(idleproc,procptr,noproc);
   end else begin
     {no candidates exist for allocation to the first processor on the list
      as soon as it becomes free. Hence some idling of the proc. will result}

     procptr:= idleproc.front^.prev; { start from the 2nd processor}
     {Go down the list till you can find a processor with a busytime greater
      than the busytime of the first processor}
     while schdr[procptr^.no].busytime <= schdr[idleproc.front^.no].busytime do
     begin
       procptr:=procptr^.prev;
     end;
     {procptr is that processor}
     temptr:=procptr^.next;
     if procptr = idleproc.back then idleproc.back := procptr^.next;
     if procptr^.prev <> nil then procptr^.prev^.next:= procptr^.next;
     procptr^.next^.prev := procptr^.prev;
     procptr^.prev := idleproc.front;
     idleproc.front^.next :=procptr;
     idleproc.front:=procptr;  {place it in front}
     {modify the busytime of the processors whose busytime was equal to the
      busytime of the old processor. They all will have to idle upto the
      busytime of the new first processor on the list }
     while temptr <> procptr do begin
       procptr:=procptr^.prev;
       schdr[procptr^.no].busytime := schdr[idleproc.front^.no].busytime;
     end;
   end;
  end;
  end;
end; {scheduler}


{***************************************************************************
                                                                    LISTOUT
 *************************************************************************** }
procedure listout(list:typelist);
{
         Print out the task list
   with tree[i] do begin
     totproctime:=totproctime+proctime;
     totnarg:=totnarg+narg;
     ptr:=arg;
     if narg > 0 then begin
       for j:=1 to narg do begin
         if procid=tree[ptr^.no].procid then begin
           nlocarcs:=nlocarcs+1;
           totcommtime:=totcommtime+ptr^.commtime;
         end else begin
           nbusarcs:=nbusarcs+1;
           totbustime:=totbustime+ptr^.bustime;
         end;
         ptr:=ptr^.next;
       end;
    end;
 end;
 totidletime:=0;
 if noproc >1 then
 for i:=1 to noproc do begin
   taskptr:=schdr[i].first;
   for j:=1 to schdr[i].notasks do begin
     if taskptr=schdr[i].first then
       totidletime:=totidletime - 0 + taskptr^.exectime.lower
     else begin if taskptr<>schdr[i].last then
       totidletime:=totidletime - taskptr^.exectime.upper
                                       + taskptr^.next^.exectime.lower

     else
       totidletime:=totidletime - schdr[i].last^.exectime.upper
                                       + tree[exitnode].tmax;

     end;
     taskptr:=taskptr^.next;
   end;
 end;
 writeln(out,'Total No. of Nodes = ',nonodes);
 writeln(out,'Total No. of Arcs = ',totnarg);
 writeln(out,'Total Bus Commtime = ',totbustime:10:3);
 writeln(out,'No. of Arcs going to other processors = ',nbusarcs);
 writeln(out,'Total Local Commtime = ',totcommtime:10:3);
 writeln(out,'No. of Arcs going to same processor = ',nlocarcs);
 writeln(out,'Total Communication Time = ',(totbustime+totcommtime):10:3);
 writeln(out,'Total Processing Time = ',totproctime:10:3);
 writeln(out,'Total Idle Time = ',totidletime:10:3);
 writeln(out,'Net Time taken by processors = ',
                      (totbustime+totcommtime+totproctime+totidletime):10:3);
end; {stats}

{***************************************************************************
               m a i n    p r o g r a m
 *************************************************************************** }
begin
  nonodes:=0;entrynode:=1;exitnode:=0;
  graphin(tree,nonodes,entrynode,exitnode); {read in the graph to be allocated}
  rewrite(out,'outgraph5');
  dataout(tree,nonodes);
  revgraph(tree,nonodes,entrynode,exitnode); {reverse the graph}
  rewrite(out,'outgraph6');
  dataout(tree,nonodes);
  evalcp(tree,exitnode); {evaluate critical paths of nodes in reversed graph}
  writeln('The critical path of the graph is ',
                                    tree[entrynode].criticalpath:10:3);

  rewrite(out,'outgraph7');
  dataout(tree,nonodes);
  setuplist(list,tree,nonodes); {create the ordered task list}
  rewrite(out,'outlist');
  listout(list);                   {print the initial task list}
  initproclist(idleproc,noproc,schdr); {initialize the processor list}
  scheduler(idleproc,schdr,tree,nonodes,noproc);
                                         {run the algoritm for allocation}

  rewrite(out,'outgraph8');
  dataout(tree,nonodes);
  rewrite(out,'outschdr');
  schdrout(schdr,noproc);  {print the final allocation}
  if noproc >1 then writeln('The response time for ',noproc,' processors is ',
                               tree[exitnode].tmax:10:3);

  rewrite(out,'outstat');
  stats(tree,nonodes,schdr,noproc,exitnode);  {print statistics on allocation}
end.
```