DISCUSSION PAPER NO. 605

HOW A NETWORK OF PROCESSORS CAN SCHEDULE ITS WORK

Stanley Reiter*

Center for Mathematical Studies in
Economics and Management Science
Northwestern University
Evanston, Illinois 60201

and

The Institute for Mathematics and its Applications
University of Minnesota
Minneapolis, Minnesota 55455

April 1984

*This research supported by NSF grant #IST 8314 504

How a Network of Processors can Schedule its Work

By Stanley Reiter

Consider a collection of processors, capable of communicating with each other and so forming a network. This network is confronted with computational tasks arriving as a flow in time, requiring different computational steps and procedures, and bearing different degrees of urgency. A task consists of a collection of operations each of which has a program and prescribes data such that the execution of the program on that data carries out the operation. The data may, of course, be the outputs of other operations. This collection of operations has a structure of precedence relations, expressing constraints on the order in which operations may be carried out. Some operations may be carried out at the same time, some must await the results of other operations before they can be performed.

The problem addressed in this paper is to design a method by which a network of processors confronted with a flow of tasks may distribute the computing to be done among the processors so as to make effective use of them to perform the required computations. The method, and its variants, presented in this paper gives weight to the objectives of carrying out the prescribed tasks in short time, and to the relative urgencies associated with those tasks. This problem is reminiscent of the problem of scheduling the flow of jobs through a machine shop. The methods presented here are adapted from a method developed for that problem which were described in [1].

One might naturally ask that the network distribute its tasks optimally, in some sense. This presents difficulties. What is to be the criterion of optimality? What is to be the time horizon over which optimization takes place? If it is a finite horizon, then boundary conditions are likely to play a role, because in this sort of problem present decisions effect the options available in the future. Statistical approaches are also conceivable. I do not explore these here. However, I suspect that the main difficulty with optimization lies in the fact that the network itself would be doing the optimizing. This means that time and capacity used to calculate a better schedule is time and capacity not available to work on the flow of computational tasks. Then full optimization would require that the value of foregone computing be weighed against the value of the improvement of scheduling resulting from the diversion of time and capacity from computing to scheduling. In turn, the time and capacity spent in doing that weighing must itself be evaluated to decide whether it is worthwhile to carry out that comparison. This leads to an infinite regress. Perhaps this regress can be cut off by some sort of fixed point property, making optimization a possibility. In any case, to demand that a network distribute its workload optimally leads to very difficult and complex problems. One common theoretical approach is to limit the problem by considering an artifically simplified and therefore manageable optimization problem. Another is to try to take into account more of the important considerations of the full problem, but to accept "sufficiently good" performance characteristics, rather than demanding "optimal" scheduling. This is the approach I take here. It should become evident that the methods presented here are designed for a large network working on a large number of tasks.

Two versions of a common basic method are described in this paper. They each permit a network of processors to schedule in a decentralized fashion the work presented to it by a flow of tasks, that is, to schedule the work without having a central decision - maker who makes the required decisions. The method described below distributes decision-making among the processors in a way that depends on the flow of work in time. Each processor, when it is free to do something, decides what to do next.

The procedure described in this paper leads to coordinated decisions with some desirable properties. The two variants of the method presented here differ in that the second emphasizes performance of the required tasks in shorter time, but involves use of more information about the operations to be carried out, and exposes more of the network to the consequences of failures. (Failure here refers to events which stop or delay processing. Failures which could generate false information are not considered here.) The methods seem to me to be quite robust under loss of parts of the network, allowing the surviving processors to proceed with the remaining work, and to accommodate the return of previously non-functional processors with no disruption.

Tasks. A computing task consists of one or more operations. An operation is a computation to be performed by a single processor. An operation has associated with it a program and possibly data. The execution of the program on those data carries out the operation. An operation, while it is the elementary unit of the method described here, may consist of a sequence of steps, perhaps repetitive, for example to calculate the square of each of n numbers $x_1 \ldots x_n$. Because the processors are not assumed to be capable of parallel calculations, this calculation would be performed by first calculating x_1^2, then x_2^2, and so on. If the squaring of each of the n numbers x_1 were taken to be a single operation then the n squaring operations might be carried out in parallel by different processors. The structuring of tasks into operations is taken as given.

The description of an operation includes information which identifies its program and data, each of which may be the result of other operations carried out previously. I will suppress formal reference to the program and data associated with an operation. I assume that when a processor finds an operation to be performed, it also finds information allowing it to retrieve the associated data and program. The labelling required to enable processors to retrieve needed information is not difficult to supply, and the method of scheduling ensures that processors consider operations only when data needed from other operations will in fact be available.

The collection of operations that make up a task has a structure. Certain operations may require that others preceed them, because they use the results of those preceeding operations, while others may be carried out independantly in parallel. Loops are excluded, i.e., there are no distinct operations i and j such that i preceeds j and j preceeds i. Conditional precedence relations are permitted. An example of this is given in Table 1'.

Tasks are most conveniently thought of as self-contained entities independent of one another. But, if it is for some reason desireable to classify the work in such a way that two tasks are interrelated, then this will be expressed in terms of precedence relations among the operations of those tasks. Precedence relations among the operation of a task may be expressed by means of a graph. A simple example follows.

Task 1

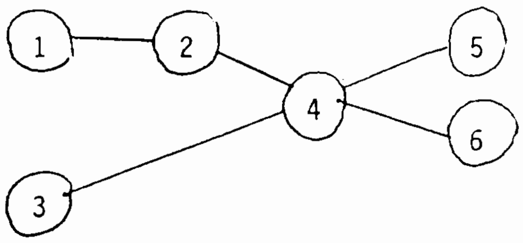


Figure 1.

Task 1 consists of 6 operations, designated 1 ,..., 6 . Operations 1 and 3 have no preceeding operations, operation 2 requires that 1 should have been performed before it can be carried out; operation 4 requires both 2 and 3 to have been carried out before it can be executed, and operations 5 and 6 produce the final results of the task and require the output of operation 4 before they can be carried out. They may be carried out in parallel, as can operations 1 and 3, or 2 and 3

This graph may be described as follows.

Table 1

task number	operation number	number preceeding operations	number of succeeding operation	preceeding operations number	succeeding operations number
1	1	0	1	-	2
1	2	1	1	1	4
1	3	0	1	-	4
1	4	2	2	2 3	5 6
1	5	1	0	4	-
1	6	1	0	4	-

The first column of Table 1 shows the task number, in this case the same for all operations. The second column shows the operation number. The third shows the number of operations which immediately preceed the given one in the graph of Task 1. This is the initial value of the <u>scheduling index</u>, which is adjusted dynamically in the course of scheduling. The third column shows the number of immediate successors. The fourth and fifth columns show the identity of the operations which immediately preceed and immediately follow (resp.) the given one.

The description of an operation will also include the processing time require to carry out that operation. One variant of our method of scheduling uses this information, another does not. This processing time may depend on the processor that carries out the operation. I will here assume that all processors take the same time to perform a given operation. No special difficulty arises when the times differ, only the description of the procedure is more complicated.

A further refinement is to be considered. Operations 1 and 2 , for example, may each consist of a sequence of steps executed on an array of data. For example, operation 1 takes two arrays $x_1 \ldots x_n$, $y_1, \ldots, y_n$ and produces $x_1 + y_1, \ldots, x_n + y_n$, while operation 2 takes $x_1 + y_1, \ldots, x_n + y_n$ and produces $(x_1 + y_1)^2, \ldots, (x_n + y_n)^2$. Suppose the time required to perform an addition is 1 unit. Then the total time needed to execute operation 1 is n units. Suppose further that it takes 1 unit of time to form z_1^2 given z_1. Then operation 2 also takes n units. The total time required for the two operations is then at least 2n units. However, if the results of operation 1 were made available to operation 2 piece by piece, the two operations could be carried out in n+1 units, operation 2 receiving $(x_1 + y_1)$ one unit after the start of operation 1 and producing $(x_1 + y_1)^2$ one unit of time later. Similarly for the others, receiving $(x_n + y_n)$ n units of time after the start of operation, and producing $(x_n + y_n)^2$ n+1 units of time after the start of the first operation. I shall refer to this mode of functioning as overlapping of operations.

Each task also carries with it information permitting the calculation of a priority number, which may be any function of the information associated with a task, including such additional information as the time it is desired that the task be completed. The priority number may be used to decide the order in which operations of the various tasks are carried out. The priority numbers of tasks may be revised dynamically in the course of scheduling.

Processors: Processors are devices capable of executing the programs associated with the operations which make up tasks. I assume that processors can communicate with one another for the purpose of exchanging information about operations, including programs and data, and to coordinate these activities.

It is convenient to describe the network of processors as if it had a central memory containing the relevant information about tasks and operations, but this is for convenience only. Tasks may enter the network through a variety of channels and memory may be distributed in the network. What is required is that processors be able to communicate in such a way that the information exchanged via the central memory in the description I give here can be exchanged directly among the processors when there is no central memory without interrupting operations being carried out on the processors being queried. It would therefore be desireable to have processors that are to this extent capable of a degree of parallel operation, i.e., they can communicate from memory and compute at the same time.*

I shall describe a procedure by which the processors of the network, confronted with a collection of tasks to be performed, schedule the execution of the operations of those tasks among themselves through time. New tasks may enter the network as processing goes on and old ones are completed. I describe first a procedure that regards operations as elementary units without internal structure and does not seek the benefits of overlapping. This procedure does not plan ahead in any way, hence, it does not require information about processing times of operations.

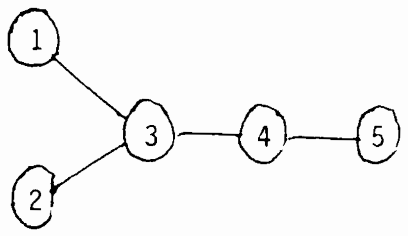
Scheduling I. At any time, the processors of the network are busy with operations. Suppose at time t some of them complete their current work and become

*) This is not strictly speaking necessary. Processors with memories could operate like time-sharing machines. But the point of parallel processing is to speed-up computations. It would therefore be better to have processors of the type described.

free. The free processors decide which of them is "first", say the one with lowest identifying number. At time t there is a collection of operations, all of whose preceeding operations have been completed. These are _ready - to - go_. The scheduling index for such an operation is "0". The first processor free at t selects an operation with highest priority from among those ready - to - go at t. If there are several such, the tie is broken, say, by choosing the operation among them with lowest identifying number. Suppose operation i is selected. The processor involved changes the scheduling index of operation i from "0" to "*", indicating the operation is in process, and executes operation i starting at t. When the operation is completed, say at time T, the processor changes the scheduling index of i from "*" to "-1", and reduces by one unit the scheduling index of every operation which is an immediate successor of i. At each time an operation is completed, i.e. when a processor seeks a new operation to carry out, the priorities associated with tasks and operations may be updated to reflect changes in the variables on which the priority indicator depends. This updating might be done by the first free processor itself.

The way this procedure works can be made clear by applying it to a simple example. In this example we have a network of three identical processors P1, P2, P3, confronted with two tasks. The first is Task 1, whose graph is shown in Figure 1., the other, Task 2, has the graph shown in Figure 2.

<u>Task 2</u>



<u>Figure 2</u>

Combining Table 1 with a similar table for Task 2, Table 2 summarizes the precedence relations for both tasks. The operations have been renumbered in sequence so that each operation has an identifying number, operation 2 of Task 2 now has the number 8.

TABLE 2

task number	operation number	number preceeding operations	number of succeeding operation	preceeding operations number(s)	succeeding operations number(s)
1	1	0	1	-	2
1	2	1	1	1	4
1	3	0	1	-	4
1	4	2	2	2 , 3	5 , 6
1	5	1	0	4	-
1	6	1	0	4	-
2	7	0	1	-	9
2	8	0	1	-	9
2	9	2	1	7 , 8	10
2	10	1	1	9	11
2	11	1	0	10	-

The information from Table 2 used for scheduling by the first method is extracted and shown in Table 3. Table 3 also has columns for the start and finish times of operations on the three available processors. In this example I assume that Task 1 has higher priority, and that this remains unchanged throughout.

(*) If there were precedence relations between Tasks 1 and 2, say, that operation 9 (Task 2) must be completed before operation 5 (Task 1) can be carried out, there would just be different numbers in the various columns of Table 3. Specifically in the row for operation 9 the column for number of succeeding operations would show "2" rather than "1" and the column for succeeding operations number(s) would list " 10, 5 " instead of just " 10 " , while the column for number of preceding operations would show " 2 " in place of " 1 " in the row corresponding to operation number 5 , and the column for preceeding operations number(s) would show " 4, 9 " instead of " 4 " in that row.

TABLE 3

task number	operation number	scheduling index	succeeding operation(s)	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
1	1	0	2	1						
1	2	1	4	2						
1	3	0	4	1						
1	4	2	5,6	1						
1	5	1	-	2						
1	6	1	-	2						
2	7	0	9	1						
2	8	0	9	2						
2	9	2	10	1						
2	10	1	11	1						
2	11	1	-	1						

In Table 3 the column labelled "scheduling index" has the initial values given by the "number of preceeding operations" shown in Table 2 for each operation. The scheduling index is adjusted as work is done to show the number of operations preceeding a given one yet to be completed. When that index has the value, 0, it signifies that all preceeding operations have been completed and the operation in question is ready-to-go. Processing times have been included in Table 3 to make it easier to follow; they are not used by this procedure.

Suppose all the processors become free at time t=0, and that the rule for deciding which is the first processor is that the one which became free earliest is first, and in case of ties the one with lowest identifying number among those tied is first. So, processor P1 is the first free processor at t=0. It finds operations with scheduling index, 0, namely operations 1, 3, 7, 8. Since Task 1 has higher priority it selects from 1, and 3, and according to the rule for breaking ties, operation 1 is selected. Then the row of Table 3 corresponding to operation number 1 is changed as follows:

task number	operation number	scheduling index	succeeding operation	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
1	1	~~0~~ *	2	1	0					

The scheduling index is changed from 0 to *, indicating that operation 1 is in process, and the blank entry in the P1 start column is replaced by 0, indicating that operation 1 is running on P1 starting at t=0.

Processor P1 informs the others that it is no longer free and P2 is now the first free processor. It finds three operations in Table 3, as modified by P1, with scheduling index, 0, namely operations 3, 7, 8 . Since Task 1 has higher priority, P2 selects 3. The row for operation 3 now appears as follows.

task number	operation number	scheduling index	succeeding operation	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
1	3	~~0~~ *	4	1			0			

The time taken by processor P1 to select the operation it runs is here assumed to be zero. Generally this would take some time, during which processor P2 (and P3) would have to wait. By assuming that this selection is done instantaneously P2 can also start at t=0. It would not change anything essential if we recognized that the communication and selection process takes time.

Processor P3 is now the first free processor, also at t=0 , it finds operations 7, 8 ready-to-go, and according to the rule for breaking ties, selects 7.

The row of Table 3 corresponding to operation 7 now appears as follows.

task number	operation number	scheduling index	succeeding operation	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
2	7	~~0~~ *	9	1					0	

Now all processors are occupied, and if all goes well, become free at t=1, having completed the selected operations. At t=1 processor P1 is the first free processor. It "reports" that operation 1 is completed by changing the scheduling index of every operation immediately succeeding operation 1. This results in changing two rows of Table 3, the row for operation 1 and that for operation 2, as follows.

task number	operation number	scheduling index	succeeding operation	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
1	1	~~0~~ ~~*~~ -1	2	1	0	1				
1	2	~~1~~ 0	4	2						

Processor P1, then proceeds to select a new operation to perform. There are now two operations ready-to-go. These are operations 2 and 8. P1 selects operation 2, and row 2 appears as follows

task number	operation number	scheduling index	succeeding operation	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
1	2	~~1~~ ~~0~~ *	4	2	1					

Then P2 becomes the first free processor and alters the rows of operations 3 and 4 as follows.

task	opera-tion	sched-uling	succeed-ing	process-ing	P1		P2		P3	
number	number	index	operation	time	start	finish	start	finish	start	finish
1	3	~~0~~ ~~1~~ -1	4	1			0	1		
1	4	~~2~~ 1	5,6	1						

Processor P2 then finds only operation 8 ready-to-go and selects it, changing the row corresponding to 8 to,

task	opera-tion	sched-uling	succeed-ing	process-ing	P1		P2		P3	
number	number	index	operation	time	start	finish	start	finish	start	finish
2	8	~~0~~ *	9	2			1			

Processor 3 is now the first free processor and it alters rows 7 and 9 as follows.

task	opera-tion	sched-uling	succeed-ing	process-ing	P1		P2		P3	
number	number	index	operation	time	start	finish	start	finish	start	finish
2	7	~~0~~ ~~1~~ -1	9	1					0	1
2	9	~~2~~ 1	10	2						

At t=1 processor P3 finds no operations ready-to-go, and so waits.

It would be possible to split into two phases what happens when a processor becomes free at t . The first phase would have each processor that becomes free at t report the completion of the operation it was working on and update the relevant scheduling indexes. In the second phase the processors would in turn select the next operation from among those ready-to-go at t.

TABLE 4

task number	operation number	scheduling index	succeeding operation	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish
1	1	Ø ƒ -1	2	1	0	1				
1	2	~~1~~ Ø ƒ -1	4	2	1	3				
							0	1		
1	3	Ø ƒ -1	4	1						
1	4	~~2~~ ~~1~~ Ø ƒ -1	5,6	1	3	4				
1	5	~~2~~ Ø ƒ -1	-	2	4	6				
1	6	~~1~~ Ø ƒ -~~1~~	-	2					4	6
2	7	Ø ƒ -1	9	1					0	1
2	8	Ø ƒ -1	9	2			1	3		
2	9	~~2~~ ~~1~~ Ø ƒ -1	10	1			3	4		
2	10	~~1~~ Ø ƒ -1	11	1			4	5		
2	11	~~1~~ Ø ƒ -1	-	1			5	6		

Table 4 shows the history of this scheduling method over time and shows the way the processors would distribute and perform the tasks using the single phase procedure. Figure 3 shows what the processors end up doing over time.

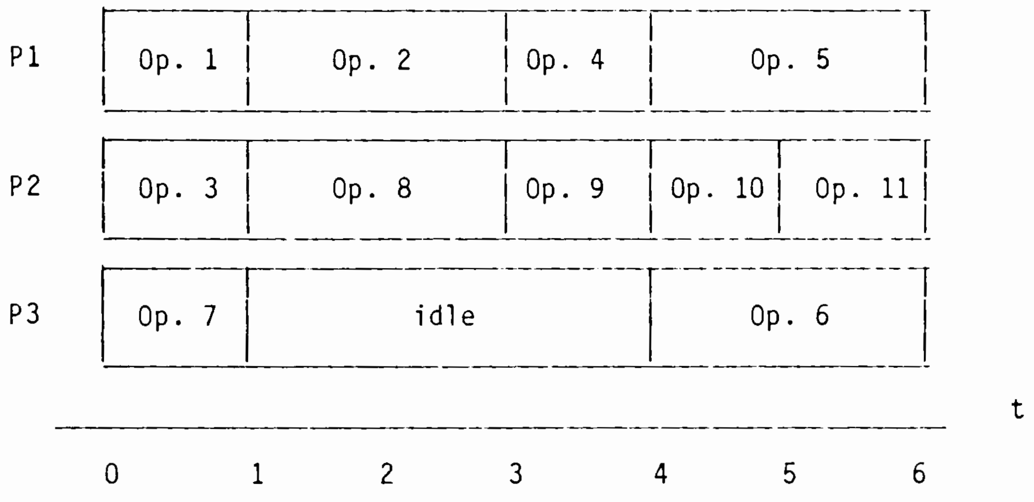


Figure 3

Use of the two phase procedure in this example, with the given priorities, would result in a redistribution of the work, (for example, P3 would perform operation 4 instead of P1), but no change in the completion times of the two tasks, (in this example).

If we consider the same example with priorities changed so that Task 2 has higher priority than Task 1, then the schedule shown in Figure 4 results.

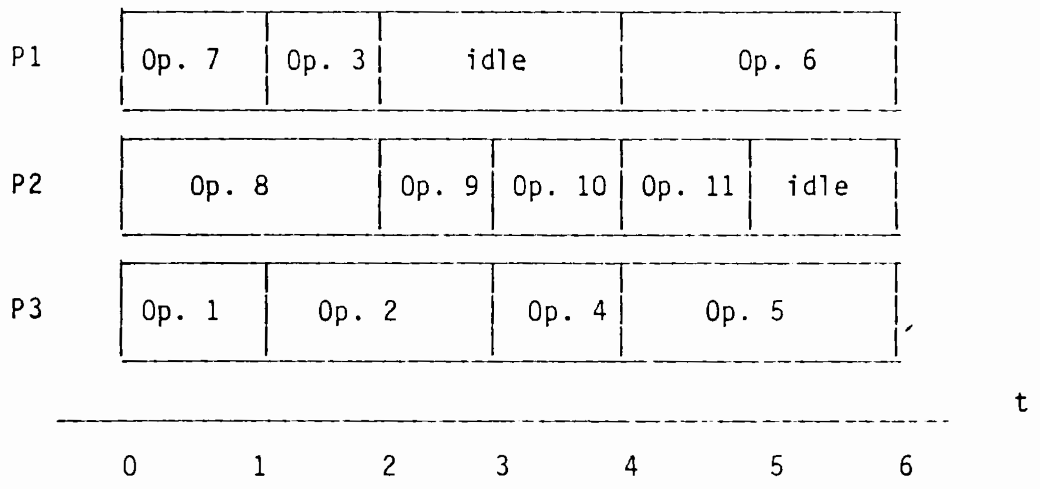


Figure 4

According to this schedule Task 2 is completed at t=5 while Task 1 still finishes at t=6.

Other distributions of the work are possible, but none with earlier completion times. For example, the principle of selecting operations with shortest processing times first leads to a schedule essentially the same as that in Figure 3. Of course, use of a different priority rule might have resulted in a different schedule for the same tasks.

A task may contain operations whose precedence relations have a conditional character. For example, operation 4 in Task 1 might require <u>either one</u> of the two preceeding operations, 2, and 3, to be completed before it can be carried out. In that case the coding of precedence relations, as in Table 1, would show in the row for operations 2, 3 and 4

Table 1'

task number	operation number	number preceeding operations	number of succeeding operation	preceeding operations number	succeeding operations number
1	2	1	1	1	4
1	3	0	1	-	4
1	4	1	2	2 or 3	5,6

It is easy to verify that the scheduling method described in connection with Table 4 are unaffected by the presence of this sort of precedence relation.

Another important possibility is that of conditional branching in the graph of a task. Using Task 1 as an example, suppose, that whenever the result of operation 4 is, say, A, then operation 5 is to be carried out, but not operation 6, while whenever the result is other than A, then operation 6 is to be carried out, but not 5. Then, the rows for operations 4, 5, and 6 in Table 1 are replaced by,

Table 1"

task number	operation number	number preceeding operations	number of succeeding operation	preceeding operations number	succeeding operations number
1	4	2	1	2,3	5 if A 6 if not A
1	5	1	0	4	-
1	6	1	0	4	-

In the case of such an operation as 4, operation 5 becomes the immediate successor of 4 if and only if the result of 4 is A, otherwise operation 6 is the successor. The scheduling index is revised as described in Table 4.

Table 5 shows the scheduling process applied to the example in which operation 4 of Task 1 has one immediate predecessor, which may be either operation 2 or 3, and has operations 5 or 6 as immediate successor conditional upon the outcome of operation 4. For definiteness the schedule calculated in Table 5 is based on the assumption that the outcome of operation 4 is A.

TABLE 5

task number	operation number	scheduling index	number succeeding operation	preceeding operation number	succeeding operation number	processing time	P1 start	P1 finish	P2 start	P2 finish	P3 start	P3 finish	1>2
1	1	~~0~~ * -1	1	-	2	1	0	1					
1	2	~~1~~ ~~0~~ ~~*~~ -1	1	1	4	2	1	3					
2	3	~~0~~ ~~*~~ -1	1	2	4	1			0	1			
1	4	~~1~~ 0 ~~*~~ -~~1~~ -2	1	2 or 3	5/A, 6/~A	1			1	2 (outcome A)			
1	5	~~1~~ 0 ~~*~~ -1	0	4	-	2			2	4			
1	6	~~1~~ ~~0~~ ~~*~~ -1	0	4	-	2							
2	7	~~0~~ ~~*~~ -1	1	-	9	1							
2	8	~~0~~ ~~*~~ -1	1	7	9	2					0	1	
2	9	~~2~~ ~~1~~ 0 ~~*~~ -1	1	7 or 8	10	1					1	3	
2	10	~~1~~ ~~0~~ * -1	1	9	11	1					3	4	
2	11	~~1~~ 0 * -1	1	10	-	1					5	6	

Figure 5 shows the schedule.

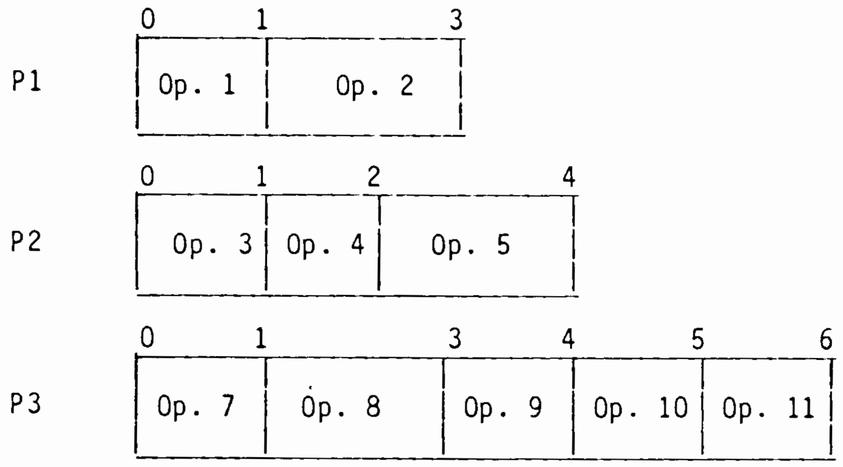


Figure 5

It should be evident that the arrival of new tasks would merely add to the list of operation; no change in the method of scheduling is needed to cope with that.

Furthermore, if there were failures in the network, the surviving processors could continue scheduling operations. Those operations which are subsequent in a task graph to uncompleted operations would remain stalled in their status at the time of failure. If a processor that failed during the execution of a scheduled operation returned to full activity and completed its scheduled operation with some delay, the system would simply pick up the situation as it actually existed at that time, and continue scheduling the remaining operations in the usual way. Depending on the priority rule used, the delay might result in higher priority for the remaining operations of the tasks delayed.

The procedure described in Tables 3 and 4, does not require knowledge of the time needed to carry out any operation. Moreover the method of scheduling described there does not involve anticipating anything. It proceeds from one event to another, making decisions on the basis of the then prevailing reality. Because no foresight is involved, it may be expected to be robust under the disturbances caused by unforseen events, such as arrival of new tasks, breakdowns, delays and the like.

Scheduling II, Overlapping Operations:

I turn now to the question of scheduling to take advantage of overlapping of operations. It is convenient to discuss this in the context of a simple example, say, a task consisting of two operations, whose graph is

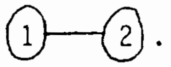


Figure 6

Operation 1 produces $x_i + y_i$ $i = 1,\ldots, n$ from data $x_1,\ldots,x_n, y_1,\ldots, y_n$; operation 2 produces z_i^2 from data z_i and the task involves setting $z_i = x_i + y_i$, so that the two operations together produce $(x_i + y_i)^2$ $i = 1,\ldots,n$ from the data $x_1,\ldots, x_n, y_1 \ldots, y_n$. Suppose each addition and multiplication takes one unit of time. I suppose that no processor is allowed to interrupt an operation in progress until it is completed, except for breakdowns. While it is conceivable that processor P1 could alternate operations 1 and 2 so as to calculate $(x_1 + y_1)^2, (x_2 + y_2)^2 \ldots$ this is ruled out here. I assume that if the hardware of processor P1 permitted that, the task would be represented as one operation.

I first describe the procedure in general terms, and then apply it to examples. According to this procedure, there is a <u>virtual schedule</u> for the network extending some predetermined time (or number of operations) into the future. This is the <u>schedule in force</u> for the network, and the all processors follow it as long as they can. If a processor, say j, should become free at a time when the schedule in force either does not prescribe an operation for that processor, (scheduled idle time may or may not be included as an operation.), or prescribes an operation that cannot be run at that time, then j asserts itself as the new scheduler, cancels the schedule and computes a new virtual schedule which then becomes the schedule in force at that time.

To take advantage of overlapping of operations, the description of an operation must be enlarged to include 1) the number N(i) of pieces of information to flow from that operation, and 2) the time p(i) required for operation i to produce one piece from its inputs, e.g., if operation i computes $(x_1 + y_i), (x_2 + y_2) \ldots (x_n + y_n)$ (in the order given) then $N(i) = n$ and the time required to carry on the operation is $p(i) \cdot N(i)$.

The following dictionary of symbols will be useful.

i -	Operation number
$t_1(i)$ -	Finish time of the first piece in operation i
$T(i)$ -	Finish time of the last piece in operation i
$N(i)$ -	Number of pieces in operation i.
$\tau(k)$ -	Earliest time at which processor k becomes free.
$\hat{\tau}(i)$ -	$\min_k \tau(k) \equiv$ earliest time at which a processor becomes free for operation i.
$e(i)$ -	Earliest potential start time of operation i, defined for operations that are ready-to-go.
$p(i)$ -	Processing time per piece in operation i.
$s(i)$ -	Scheduled start time of operation i.
$T(i) \equiv S(i)$ -	Scheduled Finish time of operation i.
$\pi(i)$ -	Set of operations of which i is an immediate successor.

At any stage of the process each operation with scheduling index 0 has an <u>earliest possible start time</u>, calculated as follows. For simplicity of exposition I consider first the case of an operation, i, having just one immediate preceeding operation, $i-1$.

Let $x(i)$ be the smallest non-negative number such that

$$x(i) + t_1(i-1) - T(i-1) + p(i)(N-1) > 0.$$

Note that this can be written as,

$$x(i) + p(i)(N-1) > p(i-1)(N-1).$$

Then

$$e(i) = \max\{t_1(i-1) + x(i), \hat{\tau}(i)\}$$

Suppose that at $t=0$ processor j is the first free processor. Processor j assumes the role of scheduler for the network, selecting at each step from the ready-to-go operations the operation i whose earliest possible start time e(i) is a minimum and having highest priority among those whose earliest possible start time is minimum. Ties are broken as in the first method. Then

$$s(i) = e(i)$$

and

$$T(i) = s(i) + p(i)\, N(i).$$

The first processor k whose earliest free time was $\tau(k) = \hat{\tau}(i)$ is the processor on which operation i is run. Processor j now updates $\tau(k)$ to the value $\tau(k) = T(i)$, decreases by one the precedence index of operation i and of every operation which is an immediate succesor of i, and revises the earliest possible start time of every operation now ready-to-go to take account of the finish times $t_1(i)$ and T(i) and therefore $\hat{\tau}(i)$ of the newly scheduled operation i.

If an operation i has more than one immediate predecessor, the earliest possible start time of i may be calculated as follows. Let $\pi(i)$ be the set of operations i' which are immediate predecessors of i.

Then define x(i) to be the smallest non-negative number such that

$$x(i) + p(i)\ (N(i)-1) > \max_{i' \epsilon \pi(i)} \{p(i)(N(i') - \frac{N(i')}{N(i)}\}$$

When N(i') = N(i) = N, this condition becomes

$$x(i) + p(i)\ (N-1) > \max_{i' \epsilon \pi(i)} \{p(i') \cdot (N-1)\}.$$

This reduces to the previous condition on x(i) when $\pi(i) = \{i-1\}$.

If the time needed to retrieve the program and data needed to run operation i is R(i), this may be accounted for by defining x(i) to be the smallest non-negative number such that

$$x(i) + t_1(i-1) + R(i) + p(i)(N-1) - T(i-1) > 0,$$

and

$$e(i) = \max\ \{t_1(i-1) + R(i) + x(i),\ \hat{\tau}(i)\},$$

for the case $\pi(i) = \{i-1\}$. The case where $\pi(i)$ is not a singleton parallels that above.

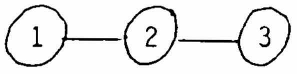
This scheduling algorithm does not keep a processor waiting in order to complete a sequence of operations more quickly by overlapping them. A simple, but perhaps too crude, way of altering this is to schedule a ready-to-go operation of higher priority if its earliest possible start time is later than that of every ready-to-go operation of lower priority by a critical amount. Let this critical number be c. At a particular stage of scheduling, among the ready-to-go operations find those with minimum earliest start time, e. Consider all ready-to-go operations, j, with higher priority than any with minimum earliest start time e, which have earliest start time $e(j) < e + c$, and schedule the highest priority operation in this class.

More sophisticated calculations could be used to decide whether to keep an available processor waiting for a higher priority operation or an operation permittting a different degree of overlapping. However such rules would have to involve comparing starting and finishing times of several operations resulting from different schedules. Such rules could get complicated rather quickly.

Examples: I turn now to some simple examples to illustrate the way this scheduling process works.

I consider first an example of a network of four processors P1, P2, P3, P4 with two tasks, whose graphs are

Task 1



Task 2

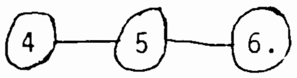


Figure 7

Suppose each operation has $N(i) = N = 10$ pieces, and that the processing time per piece of each operation is the same; $p(i) = p = 1$ for all i. Suppose that Task 1 has higher priority than Task 2, and that the critical number $c = 0$, i.e., we don't schedule idle time to wait for an operation with higher priority.

TABLE 6

task number	operation number	N(i)	P(i)	scheduling index	succeeding operation number	$t_1^{(i)}$	e(i)	P_1 start	P_1 finish	P_2 start	P_2 finish	P_3 start	P_3 finish	P_4 start	P_4 finish
1	1	10	1	0	2		0								
1	2	10	1	1	3										
1	3	10	1	1	-										
2	4	10	1	0	5		0								
2	5	10	1	1	6										
2	6	10	1	1	-										

Suppose all processors are free at t=0, and that P1 is the first free processor at t=0. Since there is no schedule in force, P1 becomes the scheduler for the network. It finds two operations ready-to-go at t=0, namely 1 and 4, and each has earliest start time e(1) = e(4) = 0. Since task 1 has higher priority, P1 selects operation 1 and schedules it on the first free processor, namely P1. The scheduling index of operation 1 is changed to -1 to indicate that the operation is scheduled. The finish time of the first piece is $t_1(1) = 1$ and of the last is T(1) = 10; processor 1 is tied up from t=0 to t=10, so that τ(P1) = 10. Operation 2 is the only successor to operation 1. Its scheduling index is reduced from 1 to 0, and, since it is now ready-to-go, its earliest start time e(2) is calculated. Since $\tau(2) = 0$, (there is a processor available at 0) and $t_1(1) + x(2) = 1$, its earliest possible start time e(2) = max{1,0} = 1.

There are now two operations ready-to-go, namely operations 2 and 4, with earliest possible start times 1 and 0, respectively. Therefore, P1 schedules operation 4 on processor P2 to start at t=0 and finish at t=10. After the prescribed updating, there are two ready-to-go operations 2, and 5, with earliest possible start times e(2) = e(5) = 1. Therefore operation 2 is the next operation scheduled by P1. TABLE 7 shows the situation after operation 2 has been scheduled.

TABLE 7

								P1		P2		P3		P4	
task number	operation number	N(i)	P(i)	scheduling index	succeeding operation number	$t_1^{(i)}$	e(i)	start	finish	start	finish	start	finish	start	finish
1	1	10	1	~~0~~ -1	2	1	0	0	10						
1	2	10	1	~~1~~ ~~0~~ -1	3	2	1					1	11		
1	3	10	1	~~1~~ 0	-		2								
2	4	10	1	~~0~~ -1	5	1	0			0	10				
2	5	10	1	~~1~~ 0	6		1								
2	6	10	1	1	-										

Operation 5 is the next operation scheduled, because its earliest possible start time is 1 compared to the earliest possible start time of operation 3, which is 2. (Operation 3 is ready-to-go after 2 has been scheduled.) After operation 5 is scheduled, the earliest possible start times of operations 3 and 6, both ready-to-go, become 10, because no processor becomes available before 10. TABLE 8 shows the history of scheduling.

TABLE 8

								P1		P2		P3		P4	
task number	operation number	N(i)	P(i)	scheduling index	succeeding operation number	$t_1^{(i)}$	e(i)	start	finish	start	finish	start	finish	start	finish
1	1	10	1	~~0~~ -1	2	1	0	0	10						
1	2	10	1	~~1~~ ~~0~~ -1	3	2	1					1	11		
1	3	10	1	~~1~~ ~~0~~ -1	-		~~2~~ 10	10	20						
2	4	10	1	~~0~~ -1	5	1	0			0	10				
2	5	10	1	~~1~~ ~~0~~ -1	6	2	1								
2	6	10	1	~~1~~ ~~0~~ -1	-		~~2~~ 10			10	20			1	11

Figure 8 shows the resulting virtual schedule.

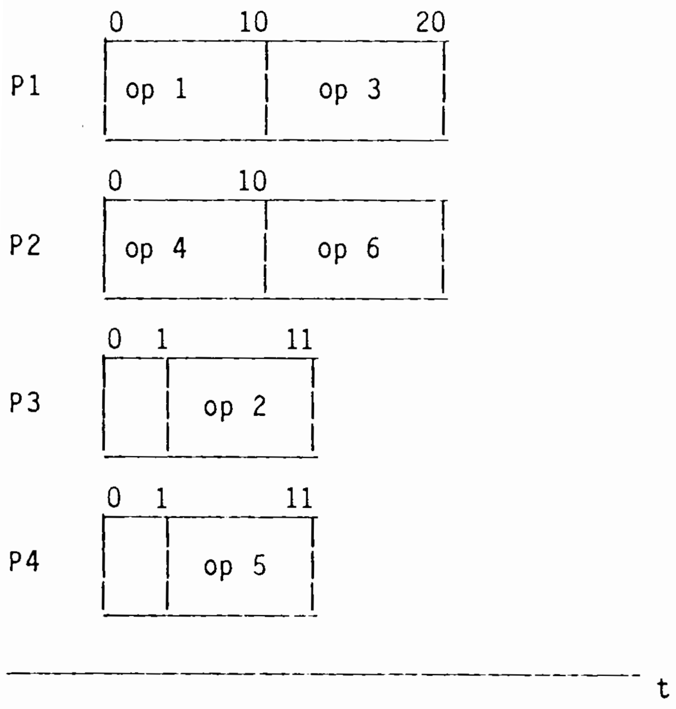


Figure 8

A slightly more complicated example for the same processor network is as follows.

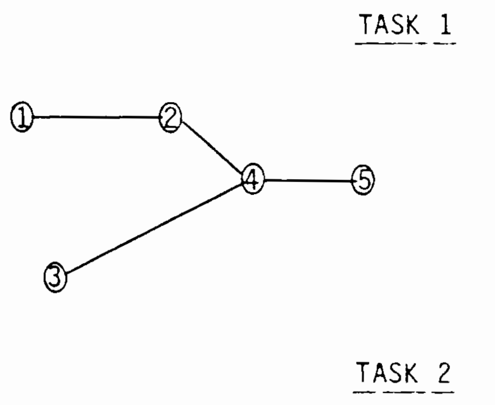


Figure 9

Suppose that for each i=1,..,8, N(i) = 10 and p(i) = 1. If Task 1 has higher priority than Task 2 and the critical number is c = 0, the schedule shown in Figure 10 results.

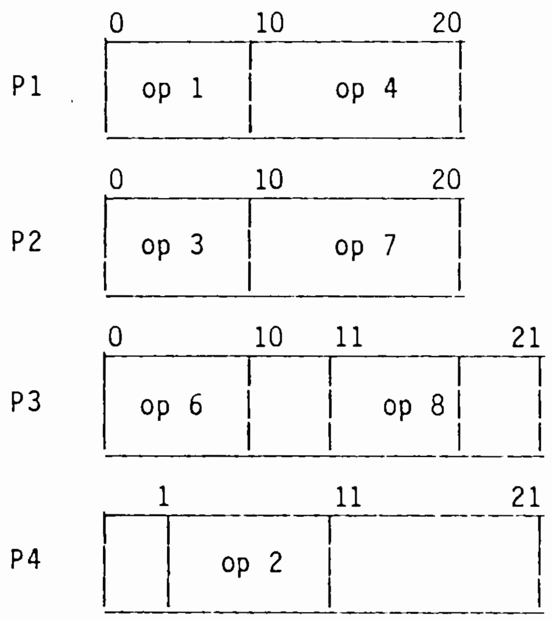


Figure 10

If task 2 has higher priority than task 1 and C=0, the schedule shown in Figure 11 results.

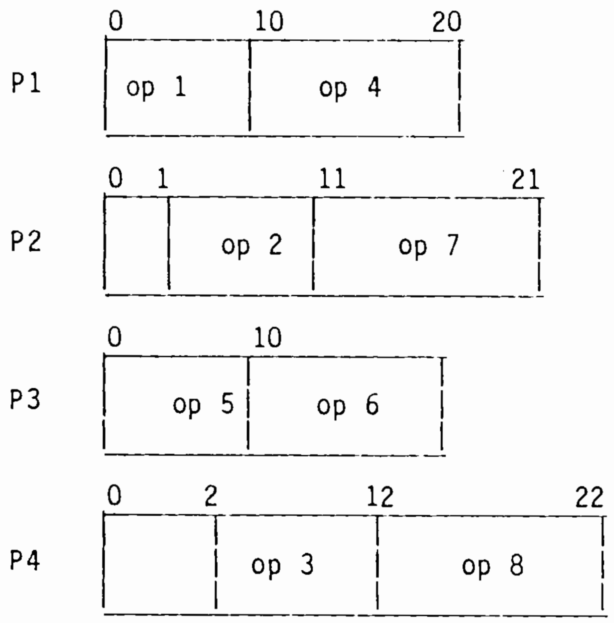


Figure 11

If Task 1 has higher priority than 2 and c=1, the schedule in Figure 12 results.

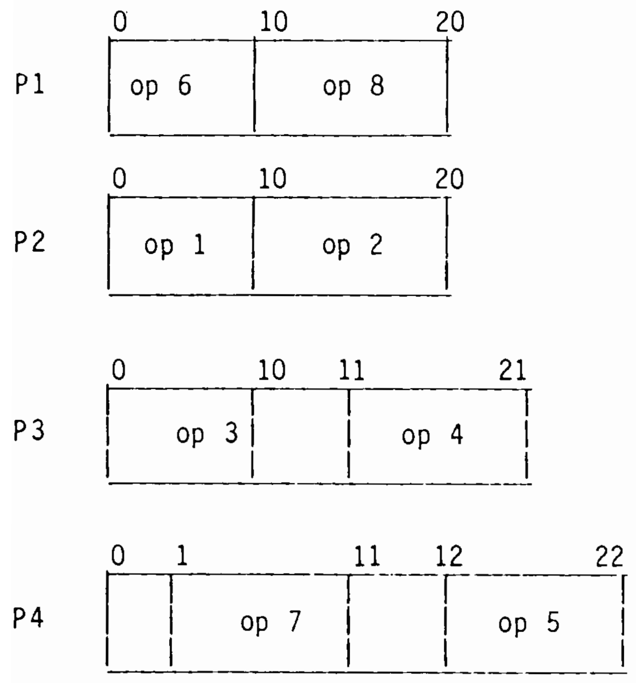


Figure 12

How a given task, or subset of them will be scheduled depends on the precise timing of the arrival of those tasks and others coming to the network during the relevant time interval, and on the priorities assigned to those tasks. Because the function used to determine priorities can depend on any of a variety of different items of information, it can be used to give effect indirectly in the scheduling process to considerations that might be quite complicated to take account of more directly. For example, if there is reason to prefer assigning certain operations to certain processors, reasons not necessarily given a priori, but generated as a result of the way processing has so far proceeded, it should be possible to adapt the methods to give effect to such preferences. An example might be to perform an operation on a processor which already has the data needed for that operation, if that processor is free at the right time.

References

[1] Reiter, S., A System for Managing Job-Shop Production, The Journal of Business of the University of Chicago. July 1966. pp. 371-393