Acta Informatica 1, 200–213 (1972)

Optimal Scheduling for Two-Processor Systems

E. G. Coffman, Jr. and R. L. Graham

Received February 8, 1971

Summary. Despite the recognized potential of multiprocessing little is known concerning the general problem of finding efficient algorithms which compute minimal-length schedules for given computations and $m \geqq 2$ processors. In this paper we formulate a general model of computation structures and exhibit an efficient algorithm for finding optimal nonpreemptive schedules for these structures on two-processor systems. We prove that the algorithm gives optimal solutions and discuss its application to preemptive scheduling disciplines.

I. Introduction

The efficient utilization of multiprocessor systems is potentially very effective in decreasing the computation times of many programs. This can be especially important for real-time applications and for large compute-bound problems. However, the problem of finding efficient, easily implemented scheduling algorithms, particularly those whose effectiveness can be demonstrated, has proved to be difficult [1]. In this paper we present such an algorithm for the nonpreemptive scheduling of so-called computation graphs on two processors. We present a proof of its optimality and illustrate the implications the result has for preemptive scheduling.

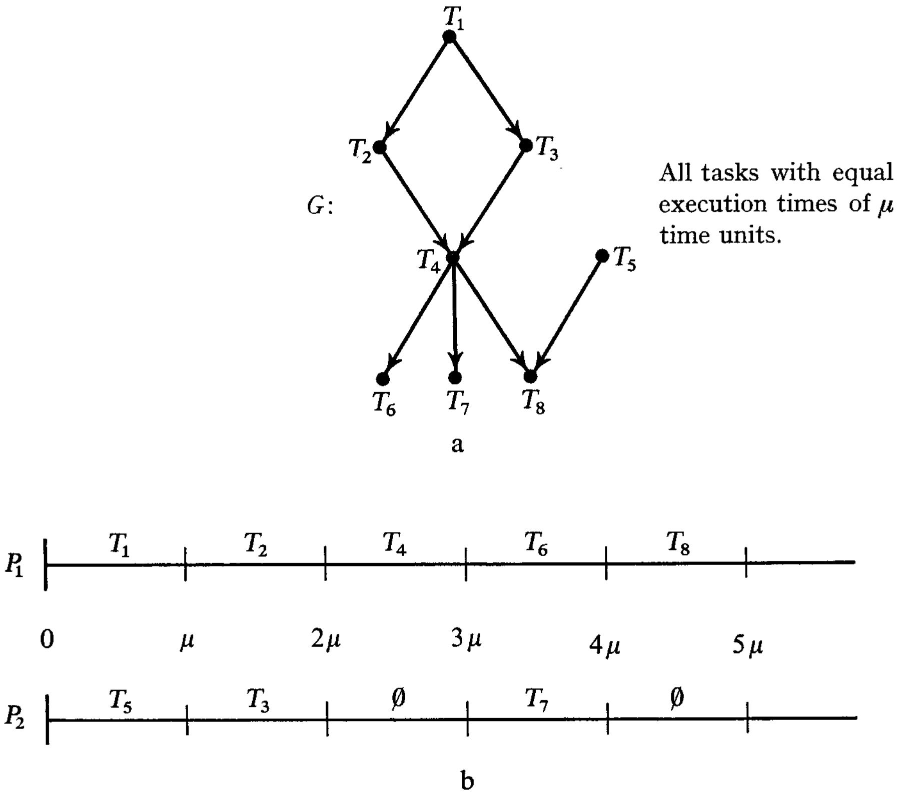
We consider systems in which there are two identical processors and assume that the computations submitted to the system are specified as a finite set G and a partial order $\prec$ on G. The elements of G will be called *tasks*. For T, $T' \in G$, if $T \prec T'$ we shall say that T is a *predecessor* of T' and that T' is a *successor* of T. In addition, if there exists no task $T'' \in G$ such that $T \prec T'' \prec T'$ then T will be called an *immediate predecessor* of T' and T' will be called an *immediate successor* of T. The set of immediate successors of T will be denoted by $S(T)$. With each computation we can associate a directed graph G whose vertices are the set of tasks*. There is a directed arc from the vertex (task) T_i to the vertex (task) T_j if and only if T_i is an immediate predecessor of T_j. An example is shown in Fig. 1a.

Informally, a schedule or assignment for a given computation is a description of the work done by each processor as a function of time. Of course the schedule must not violate** the precedence relations given by the partial order $\prec$ and we are not permitted to assign more than one processor to a task or more than one task to a processor at any time. A simple way of specifying a schedule uses a Gantt [2] chart, which consists of a time axis for each processor with intervals

* As usual, a graph and its set of vertices will be denoted by the same symbol.

** i.e., if $T_i \prec T_j$ then T_i must be completed before T_j can be begun.

marked off and labeled with the name of the task being executed. We use the symbol ∅ to denote an idle period. Fig. 1 b shows the Gantt chart of a schedule for the graph of Fig. 1 a, assuming that all tasks have the same execution time μ, and that the system consists of two processors, P_1 and P_2.



All tasks with equal execution times of μ time units.

a

b

Fig. 1. An example computation and schedule

We shall consider schedules for a graph G according to which of the tasks of G are executed by P_1 and P_2 in the following way. Let $L = (T_1, T_2, \ldots, T_r)$ denote some permutation of the tasks of G. L will be called a *list* for G. Initially, at time $t = 0$, processor P_1 begins to execute the first task in L which has no predecessor. We say that a task T is *ready* at time t if at this time all the predecessors, if any, of T have already been executed. In general, at any time a processor P_j is idle, it (instantaneously) scans the list L and begins to execute that *ready task* T_i *with minimal* i which has not yet begun to be executed. Once a processor begins to execute a task T, it continues executing T for exactly *one* unit of time, after which the execution of T is completed.. That is, all tasks in G are assumed to have the same execution time, and this is taken to be of unit length. We make the convention that if both P_1 and P_2 simultaneously attempt to execute the same task, then that task is executed by P_1. Note that the schedule in Fig. 1 b has the properties imposed by the above sequencing procedure for the list $(T_1, T_2, \ldots, T_8)$. The length of time required to execute all the tasks of G using the list L will be denoted by $\omega(L)$.

Sequencing disciplines of the above type are classified as *nonpreemptive* since a task once assigned to a processor must be allowed to run to completion. However, with *preemptive* scheduling we are allowed to "interrupt" a processor, remove the currently assigned (incomplete) task, and assign another task to the processor in its place. Preempted tasks must be later reassigned until each task receives an amount of processor time equal to its execution time. In general, with the preemption capability the overall computation times of graphs can be reduced. Although the major result of this paper concerns nonpreemptive scheduling, its application to preemptive scheduling will be discussed in a later section.

The general scheduling problem for the system we have defined consists of finding an "efficient" algorithm for sequencing the tasks in a given graph so that the total execution time is minimized. Of course, the problem is finite so that it could in principle be solved by an examination of all the possibilities. However, by an "efficient" algorithm we mean an essentially nonenumerative one, e.g., one which requires a number of steps which is algebraic in the number of tasks of G as opposed to being exponential in this number. Our approach to this problem consists of defining an algorithm that produces a list L^* such that the schedule generated by L^* has minimal length, i.e., $\omega(L^*) \leqq \omega(L)$ for any list L. The case in which the graph G is a rooted directed tree, the number of processors is arbitrary, and all tasks have equal execution times was previously considered by Hu [3]. Essentially, he showed that by assigning a priority to a task T equal to the number of tasks in the longest chain from T to any terminal vertex, if tasks are executed with highest priority tasks attempted first, this results in a minimal length schedule.

In this paper we allow G to be an arbitrary acyclic directed graph, with all tasks having equal execution times, although we restrict ourselves to two-processor systems (possible extensions are discussed at the end of the paper). This case has also recently been considered by Fujii, Kasami, and Ninomiya [8, 9]. They showed that an optimal schedule can be constructed from a maximal matching for the incomparability graph of the given partial order. The best upper bound currently known [10] for the complexity of a maximal matching algorithm for a graph on n vertices is of the order of n^4, although it appears [11] that this can be reduced to n^3.

It is not difficult to verify that the algorithm we present in the following section has an order of n^2, which, in a certain sense, is best possible since up to $\frac{1}{4}n^2$ arcs are necessary in general to specify a partial order. It will also be seen that this model is applicable to those situations in which computations can be partitioned into equal sized tasks or when preemptions are allowed for tasks having arbitrary execution times.

II. The Scheduling Algorithm and Proof of its Optimality

We present below an algorithm, to be called Algorithm A, for constructing the optimal list L^*. First, we need the following definition.

We linearly order decreasing sequences of positive integers as follows. If $N=(n_1, n_2, \ldots, n_t)$ and $N'=(n'_1, n'_2, \ldots, n'_{t'})$ are decreasing sequences of positive integers (where possibly $t=0$) we shall say that $N<N'$ if either

(i) for some $i \geqq 1$, we have $n_j = n'_j$ for all j satisfying $1 \leqq j \leqq i-1$ and $n_i < n'_i$, or

(ii) $t<t'$ and $n_j = n'_j$, $1 \leqq j \leqq t$.

Let r denote the number of tasks in G. Algorithm A assigns to each task T, an integer $\alpha(T) \in \{1, 2, \ldots, r\}$. The map α is defined recursively as follows.

(a) An arbitrary task T_0 with $S(T_0)=\emptyset$ is chosen and $\alpha(T_0)$ is defined to be 1.

(b) Suppose for some $k \leqq r$, the integers $1, 2, \ldots, k-1$ have been assigned. For each task T for which α has been defined on all elements of $S(T)$, let $N(T)$ denote the decreasing sequence of integers formed by ordering the set $\{\alpha(T'): T' \in S(T)\}$. At least one of these tasks T^* must satisfy $N(T^*) \leqq N(T)$ for all such tasks T. Choose one such T^* and define $\alpha(T^*)$ to be k.

(c) We repeat the assignment in (b) until all tasks of G have been assigned some integer.

Finally, the list L^* is defined by Algorithm A to be $(U_r, U_{r-1}, \ldots, U_1)$ where $\alpha(U_k)=k$, $1 \leqq k \leqq r$.

In Fig. 2 we give an example of a graph and a list L^* produced by Algorithm A. Note that there are many lists satisfying Algorithm A since $S(T_8)=S(T_9)$, $S(T_{10})=S(T_{11})=S(T_{12})$, $S(T_3)=S(T_4)$, and $S(T_1)=S(T_2)=\emptyset$. The schedule generated by L^* is also shown in the figure.

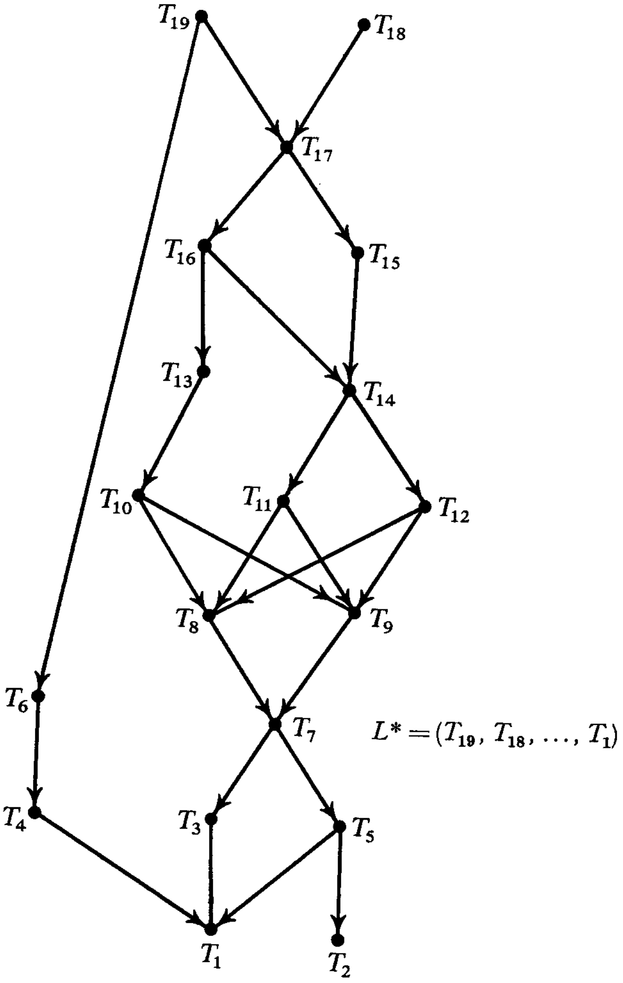
Because of the unit execution times both P_1 and P_2 become available for executing tasks at the same times. Suppose that P_1 and P_2 become available for executing new tasks at time t, and that T is the unexecuted task whose label is highest of those corresponding to tasks as yet unexecuted at time t. For all T' such that $T' \prec T$ the labeling produced by Algorithm A is such that $\alpha(T') > \alpha(T)$. Hence, at time t all predecessors of T must have been executed and T is ready to be executed. Since by the construction of L^*, P_1 and P_2 always attempt to execute the unexecuted task with the highest label, and since P_1 is assigned tasks before P_2 by convention, we see that T must be the task executed by P_1 in the unit interval beginning at time t. This establishes the following property which we put in the form of a lemma for ease of reference in the proof that L^* is optimal.

Lemma 1. Define $\tau(T)$ as the nonnegative integer representing the time at which task $T \in G$ begins execution in a schedule corresponding to the list L^* for G. If T is executed by P_1 and $\tau(T) \leqq \tau(T')$, $T \neq T'$, then $\alpha(T) > \alpha(T')$.

From the above we note also that P_1 is never idle before time $\omega(L^*)$. The principal result now follows.

Theorem 1. For a given graph G, $\omega(L^*) \leqq \omega(L)$ for all lists L.

Proof. We begin with some definitions. Suppose the tasks of G are executed using L^*. If P_2 is idle from time t to time $t+1$, we say that P_2 is executing an *empty task* $\emptyset$ and we define $\alpha(\emptyset)=0$. We recursively define tasks V_i and W_i as follows:



N.B. Tasks have been indexed so that the index of a task is equal to the label assigned by Algorithm A, i.e., $\alpha(T_j) = j$.

P_1	T_{19}	T_{17}	T_{16}	T_{14}	T_{12}	T_{10}	T_9	T_7	T_5	T_2
P_2	T_{18}	T_6	T_{15}	T_{13}	T_{11}	T_4	T_8	$\emptyset$	T_3	T_1

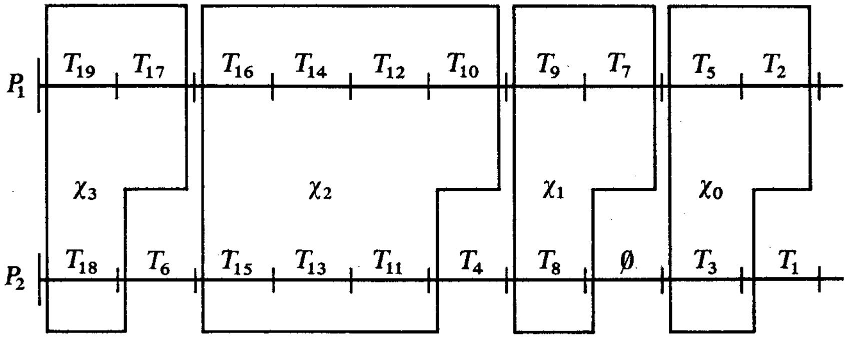
Fig. 2. Example of Algorithm A

(i) V_0 is defined to be the task executed by P_1 satisfying $\tau(V_0) = \omega(L^*) - 1$ (i.e., V_0 is the last task to be executed by P_1). Similarly W_0 is defined to be the (possibly empty) task executed by P_2 with $\tau(W_0) = \omega(L^*) - 1$.

(ii) In general, for $k \geqq 1$, W_k is defined to be the (possibly empty) task T for which $\alpha(T) < \alpha(V_{k-1})$, $\tau(T) < \tau(V_{k-1})$ and $\tau(T)$ is ***maximal***. It follows from Lemma 1 that W_k must be executed by P_2. V_k is defined to be the task executed by P_1 satisfying $\tau(V_k) = \tau(W_k)$.

If W_1 does not exist then no processor is idle before time $\omega(L^*) - 1$ and L^* is clearly optimal. Hence, we may assume that W_1 (and therefore V_1) exists. Suppose that we are only able to define W_i for $0 \leqq i \leqq m$. Define $\mathscr{X}_i$ to be the set of tasks T satisfying $\tau(V_{i+1}) < \tau(T) \leqq \tau(V_i)$ but with $T \neq W_i$, $0 \leqq i \leqq m$. Since V_{m+1} does not exist then $\mathscr{X}_m$ is the set of tasks T with $\tau(T) \leqq \tau(V_m)$ and $T \neq W_m$.

Note that the cardinality of each χ_k is odd and we can set $|\mathscr{X}_k| = 2n_k - 1$ for a positive integer n_k, $0 \leqq k \leqq m$. An example illustrating the above definitions is provided by the Gantt chart for Fig. 2. This chart is reproduced in Fig. 3 where boxes have been drawn to isolate the sets $\mathscr{X}_i$.



Note that task indices are chosen to correspond to labels, i.e., $\alpha(T_j) = j$

Fig. 3. Example for definition of $\mathscr{X}_i$

The heart of the proof of the theorem is contained in the fact that for $0 \leqq k \leqq m$, if $T \in \mathscr{X}_k$, $T' \in \mathscr{X}_{k+1}$ then $T' \prec T$. We proceed to prove this by double induction on $\tau(T)$ and $\tau(T')$. By the definition of $\mathscr{X}_k$, $T \in \mathscr{X}_k$ implies $\alpha(T) \geqq \alpha(V_k)$ and $\tau(T) \leqq \tau(V_k)$.

First, let $X \in \mathscr{X}_k$ with $\tau(X)$ minimal, i.e., $\tau(X) = \tau(V_{k+1}) + 1 = \tau(V_k) - n_k + 1$. Since

$$\alpha(V_{k+1}) > \alpha(X) \geqq \alpha(V_k) > \alpha(W_{k+1})$$

then X was called by P_2 to be executed at time $\tau(V_{k+1})$ but it was not executed. Hence, at that time some predecessor X' of X must not have been executed. Thus, $\tau(X') \geqq \tau(V_{k+1})$. But this implies $\alpha(X') > \alpha(X)$ by the definition of Algorithm A. Since X was executed at time $\tau(X)$, X' must be executed before X and

$$\tau(X') \leqq \tau(X) - 1 = \tau(V_{k+1}).$$

Therefore $\tau(X') = \tau(V_{k+1})$ and $\alpha(X') > \alpha(W_{k+1})$. There is only one possibility for X', namely, $X' = V_{k+1}$. Thus, we have $V_{k+1} \prec X$.

Next, suppose for a fixed j, $1 \leqq j < n_k$, we have shown that $X \in \mathscr{X}_k$ and $\tau(X) \leqq \tau(V_k) - n_k + j$ imply $V_{k+1} \prec X$. Let $X' \in \mathscr{X}_k$ with $\tau(X') = \tau(V_k) - n_k + j + 1$. Since

$$\alpha(V_{k+1}) > \alpha(X') \geqq \alpha(V_k) > \alpha(W_{k+1})$$

then, as before, X' was called by P_2 at time $\tau(V_{k+1})$ to be executed but it was not ready to be executed. Thus, some predecessor X'' of X' had not been executed by this time and we must have $\tau(X'') \geqq \tau(V_{k+1}) = \tau(V_k) - n_k$. Also, since $X'' \prec X'$ then $\tau(X'') \leqq \tau(X') - 1 = \tau(V_k) - n_k + j$. If $\tau(X'') = \tau(V_k) - n_k$ then since $\alpha(X'') > \alpha(X') \geqq \alpha(V_k) > \alpha(W_{k+1})$, we must have $X'' = V_{k+1}$ and we obtain $V_{k+1} \prec X'$ as required. Hence, we may assume $\tau(X'') \geqq \tau(V_k) - n_k + 1$. By the induction hypothesis, since

$$\tau(V_k) - n_k + 1 \leqq \tau(X'') \leqq \tau(V_k) - n_k + j$$

we see that $X'' \in \mathscr{X}_k$ and $V_{k+1} \prec X''$. Therefore, by the transitivity of the partial order $\prec$ we obtain $V_{k+1} \prec X'$ and the first induction step is completed. This shows that

$$V_{k+1} \prec X \quad \text{for all } X \in \mathscr{X}_k.$$

Let I_k denote the set of tasks in $\mathscr{X}_k$ which have *no predecessor in* $\mathscr{X}_k$. Since $V_{k+1} \prec X$ for all $X \in \mathscr{X}_k$ then it is not difficult to see that $S(V_{k+1}) \cap \mathscr{X}_k = I_k$.

Suppose now for some j, $0 \leqq j \leqq n_{k+1} - 2$, we have shown that if $T \in \mathscr{X}_{k+1}$ with $\tau(V_{k+1}) - j \leqq \tau(T) \leqq \tau(V_{k+1})$ then $T \prec X$ for all $X \in \mathscr{X}_k$. Let $X' \in \mathscr{X}_{k+1}$ with $\tau(X') = \tau(V_{k+1}) - j - 1$. Since $X' \in \mathscr{X}_{k+1}$ then $\alpha(X') > \alpha(V_{k+1})$. Thus, we must have $N(X') \geqq N(V_{k+1})$ where we recall that $N(X')$ is formed by taking the decreasing sequence of α-values of the immediate successors of X'. If there exists $X'' \in S(X') \cap \mathscr{X}_{k+1}$ then by the induction hypothesis, since $\tau(X'') > \tau(X') = \tau(V_{k+1}) - j - 1$, i.e., $\tau(X'') \geqq \tau(V_{k+1}) - j$, then $X'' \prec X$ for all $X \in \mathscr{X}_k$, and by transitivity, $X' \prec X$ for all $X \in \mathscr{X}_k$. Thus, we can assume that $S(X') \cap \mathscr{X}_{k+1}$ is empty. By Lemma 1, $\alpha(T) < \alpha(V_k)$ if $\tau(T) > \tau(V_k)$. Also, we see $\alpha(W_k) < \alpha(V_k)$. A moment's reflection now shows that from the definition of Algorithm A, if $N(X') \geqq N(V_{k+1})$ and X' has no successor in $\mathscr{X}_{k+1}$ then we must have $S(X') \cap \mathscr{X}_k = I_k$. This in turn implies $X' \prec X$ for all $X \in \mathscr{X}_k$. This completes the induction step and the proof that if $T \in \mathscr{X}_k$ and $T' \in \mathscr{X}_{k+1}$, $0 \leqq k \leqq m$, then $T' \prec T$.

It is now a short step to a proof of the theorem. For an arbitrary list L, *all* the tasks in $\mathscr{X}_{k+1}$ must still be executed before *any* task in $\mathscr{X}_k$ can be started. Since $\mathscr{X}_{k+1}$ consists of $2n_{k+1} - 1$ tasks then this will require at least n_{k+1} units of time. Thus, to execute G will require at least $\sum_{k=0}^{m} n_k$ units of time, no matter what list L is used. Since $\omega(L^*) = \sum_{k=0}^{m} n_k$ we have shown $\omega(L) \geqq \omega(L^*)$ and the proof is complete.

III. Remarks on the Generality of Algorithm A

At this stage one is naturally led to consider possible extensions of the preceding results. For example, one might consider the problem of algorithms for optimal lists in the case of 3 (or more) processors. The simple example in Fig. 4 shows that Algorithm A no longer always yields optimal lists.

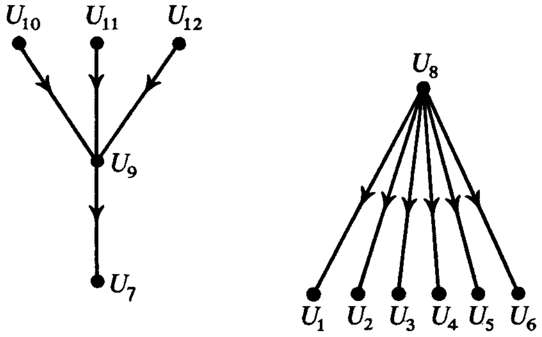
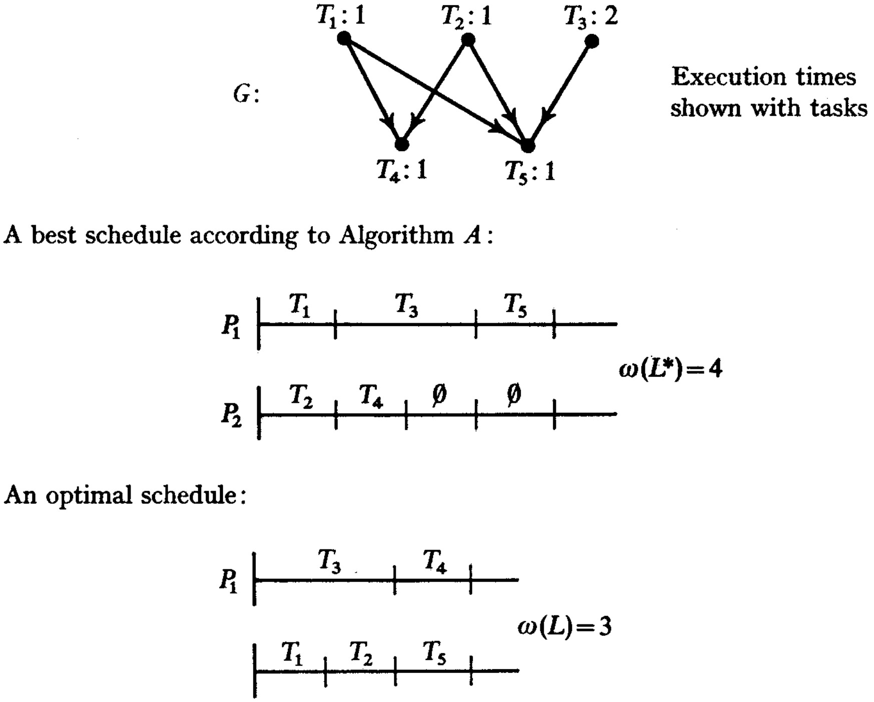


Fig. 4

One possible labeling resulting from the application of Algorithm A is $\alpha(U_k) = k$, $1 \leqq k \leqq 12$. If G is executed using the corresponding $L^* = (U_{12}, U_{11}, \ldots, U_1)$ we find that $\omega(L^*) = 5$ where, in fact, the list $L = (U_{12}, U_{11}, U_8, U_{10}, U_9, U_7, \ldots, U_1)$ yields $\omega(L) = 4$.

Similarly, one might use only 2 processors but allow the tasks to have different execution times. However, the example in Fig. 5 shows that even if we allow execution times to be either one or two units in length, then Algorithm A can fail to be optimal.



Execution times shown with tasks

A best schedule according to Algorithm A:

An optimal schedule:

Fig. 5

We remark that Algorithm A might be termed a "local" algorithm in the sense that it recursively labels a task T of G just according to the labels of the immediate successors of T. It would be interesting to know if local algorithms for optimal lists exist for the extensions mentioned.

IV. Application to Preemptive Scheduling

Apart from its theoretical significance Algorithm A of the preceding section has its principal application in the design of preemptive scheduling algorithms. To illustrate this application suppose we have an arbitrary computation graph G, whose tasks $T_1, \ldots, T_n$ have execution times $\mu_1, \ldots, \mu_n$, respectively. We assume that the μ_i's are arbitrary subject to the constraint that they be mutually commensurable*, i.e., there exists a real number w such that each task execution time can be expressed as an integral multiple of w. (Hereafter, w will be taken as the largest such number.)

In terms of w we can define a graph G_w which is obtained from G by replacing each task T_i by a chain of n_i subtasks $T_{i1}, \ldots, T_{in_i}$ having execution times of w time units where $\mu_i = n_i w$. For example, in Fig. 6a we choose $w=1$ and obtain the graph G_1 in Fig. 6b. Assuming that we have a system in which preemptions are allowed only at times $w, 2w, 3w, \ldots$, an optimal nonpreemptive schedule for G_w (as obtained by Algorithm A) can be viewed as an optimal preemptive schedule for G. In Fig. 6 we have also indicated the schedules for G and G_1 which show the improvement by a factor of 7/6 made possible by allowing preemptions at the end of each unit interval.

The question naturally arises as to whether an even finer subdivision of the tasks would lead to even shorter preemptive schedules; that is, is it beneficial to be able to preempt at more frequent intervals. In general, the answer is yes although we need not consider preemptions at intervals more frequent than one every $w/2$ time units for two processors. More precisely, it is known [4] that any preemptive schedule (preemption times completely unconstrained) for a graph G is at least as long as the optimal nonpreemptive schedule for $G_{w/2}$. In Fig. 7 we have shown $G_{w/2}$ for the graph in Fig. 6a. As shown in the figure the optimal nonpreemptive schedule for $G_{w/2}$ (and therefore the optimal preemptive schedule for G) improves over G_w and represents an improvement over the schedule for G by a factor of 14/11.

For a given graph G let ω_N and ω_P denote the lengths of the optimal nonpreemptive and preemptive schedules, respectively. For an arbitrary number, $m \geqq 1$, of processors it is known [5] that an upper bound for ω_N/ω_P is given by

$$\frac{\omega_N}{\omega_P} \leqq 2 - \frac{1}{m}.$$

By modifying an example used for illustrating multiprocessing anomalies [6] we can show that the above bound is best possible. The example is shown in

* Of course, any set of μ_i's can be approximated arbitrarily closely by mutually commensurable ones.

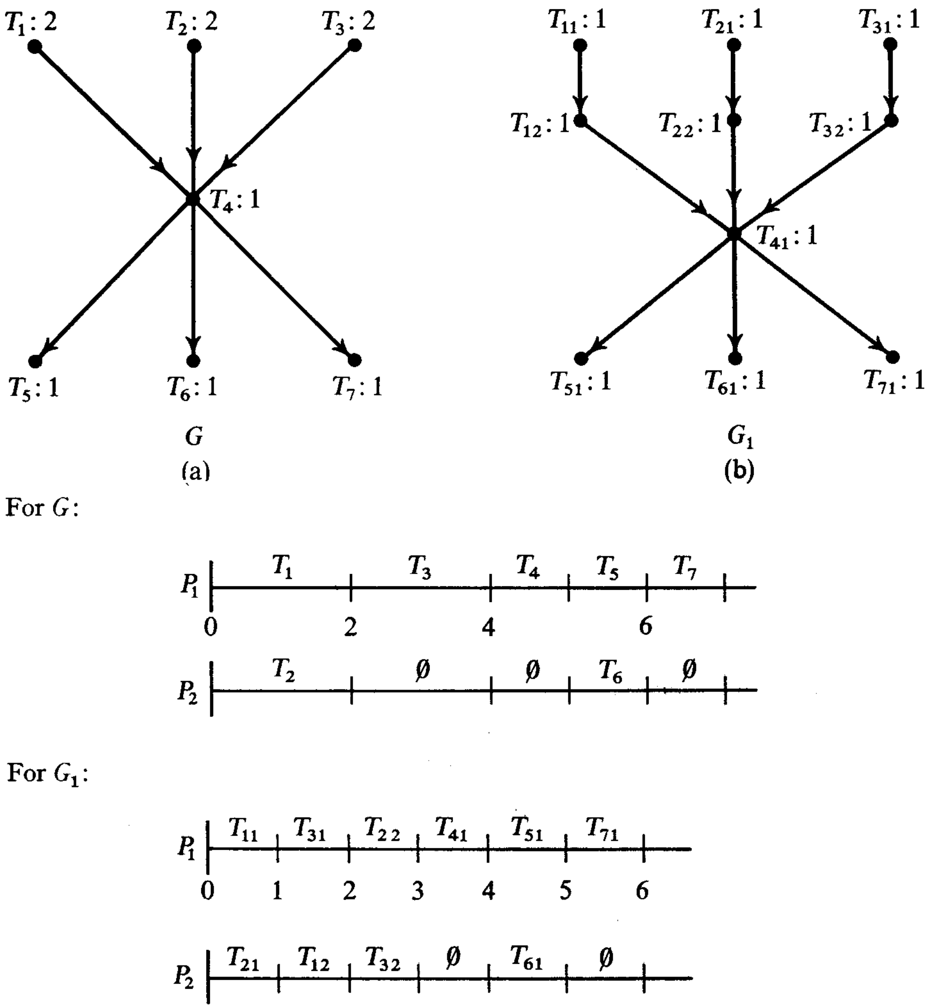


Fig. 6. Illustrating G_w

Fig. 8 along with optimal schedules. As can be verified from the figure

$$\frac{\omega_N}{\omega_P} = \frac{2m-1+\varepsilon}{m+\varepsilon}$$

so that for ε sufficiently small, the ratio approaches $2-1/m$.

When the graphs $G_{w/2}$ are large a more efficient procedure for obtaining the best preemptive schedules for two processors can be given. To present this procedure and to see its relation to Algorithm A it is convenient to introduce so-called processor-sharing disciplines. For these disciplines the m processors are considered to comprise a certain total amount of computing capability rather than being discrete units. It is assumed that this computing capability

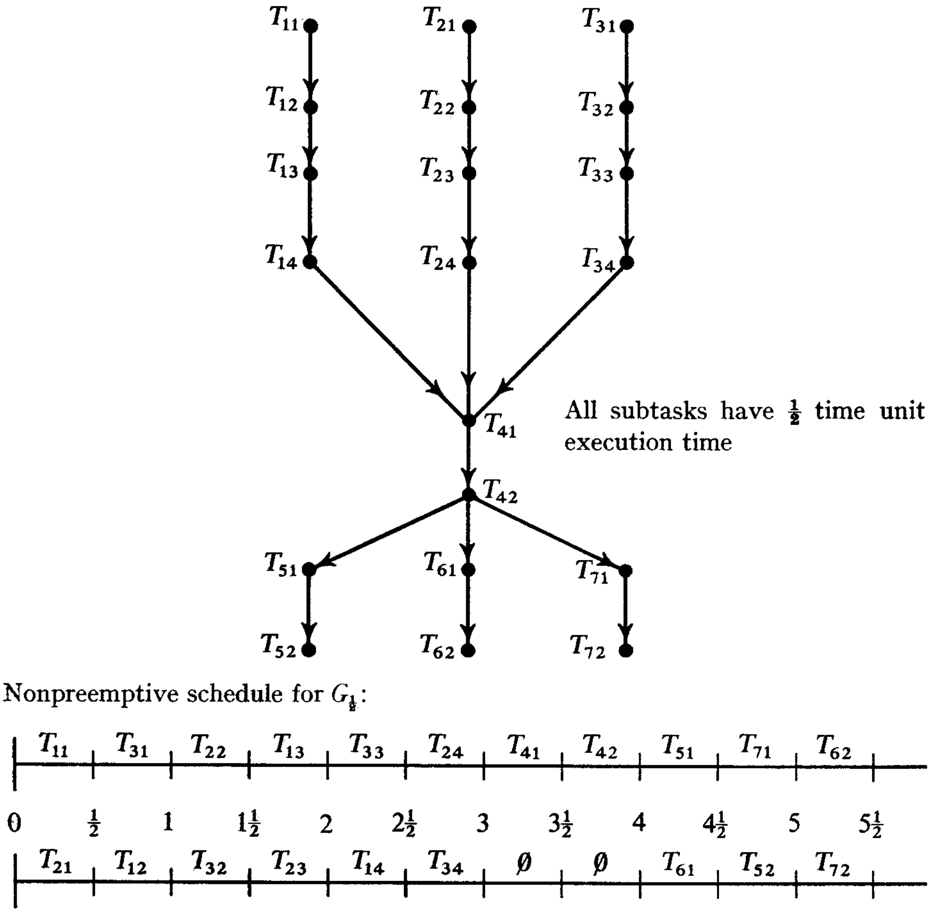

T_{11}
T_{21}
T_{31}
T_{12}
T_{22}
T_{32}
T_{13}
T_{23}
T_{33}
T_{14}
T_{24}
T_{34}
T_{41}
All subtasks have $\frac{1}{2}$ time unit execution time
T_{42}
T_{51}
T_{61}
T_{71}
T_{52}
T_{62}
T_{72}
Nonpreemptive schedule for $G_{\frac{1}{2}}$:
T_{11} T_{31} T_{22} T_{13} T_{33} T_{24} T_{41} T_{42} T_{51} T_{71} T_{62}
0 $\frac{1}{2}$ 1 $1\frac{1}{2}$ 2 $2\frac{1}{2}$ 3 $3\frac{1}{2}$ 4 $4\frac{1}{2}$ 5 $5\frac{1}{2}$
T_{21} T_{12} T_{32} T_{23} T_{14} T_{34} $\emptyset$ $\emptyset$ T_{61} T_{52} T_{72}

Fig. 7. An optimal preemptive schedule

can be assigned to tasks in any amount between 0 and the equivalent of one processor; as before a task can never be executed at a rate that exceeds that achievable on a single processor. If we assign an amount β, $0<\beta\leqq 1$ of computing capability to a task then we assume that the execution time of the task is increased by a factor of $1/\beta$. For example, if $\beta=1/2$ for a task T with execution time μ then T will take 2μ units of time to complete if it continues to be executed at this rate.

It is easily shown [4] (and illustrated below) that any schedule involving processor-sharing can be replaced by a preemptive schedule of precisely the same length. However, an optimal preemptive scheduling algorithm for two processors is most easily expressed indirectly in terms of processor-sharing. Before giving this algorithm we need the following definition.

If $T_i, T_1, T_2, \ldots, T_k, T_j$ is a path from T_i to T_j then the length of this path is defined to be $\mu_i+\sum_{l=1}^{k}\mu_l+\mu_j$. The *level* of a task T_i is defined to be the length

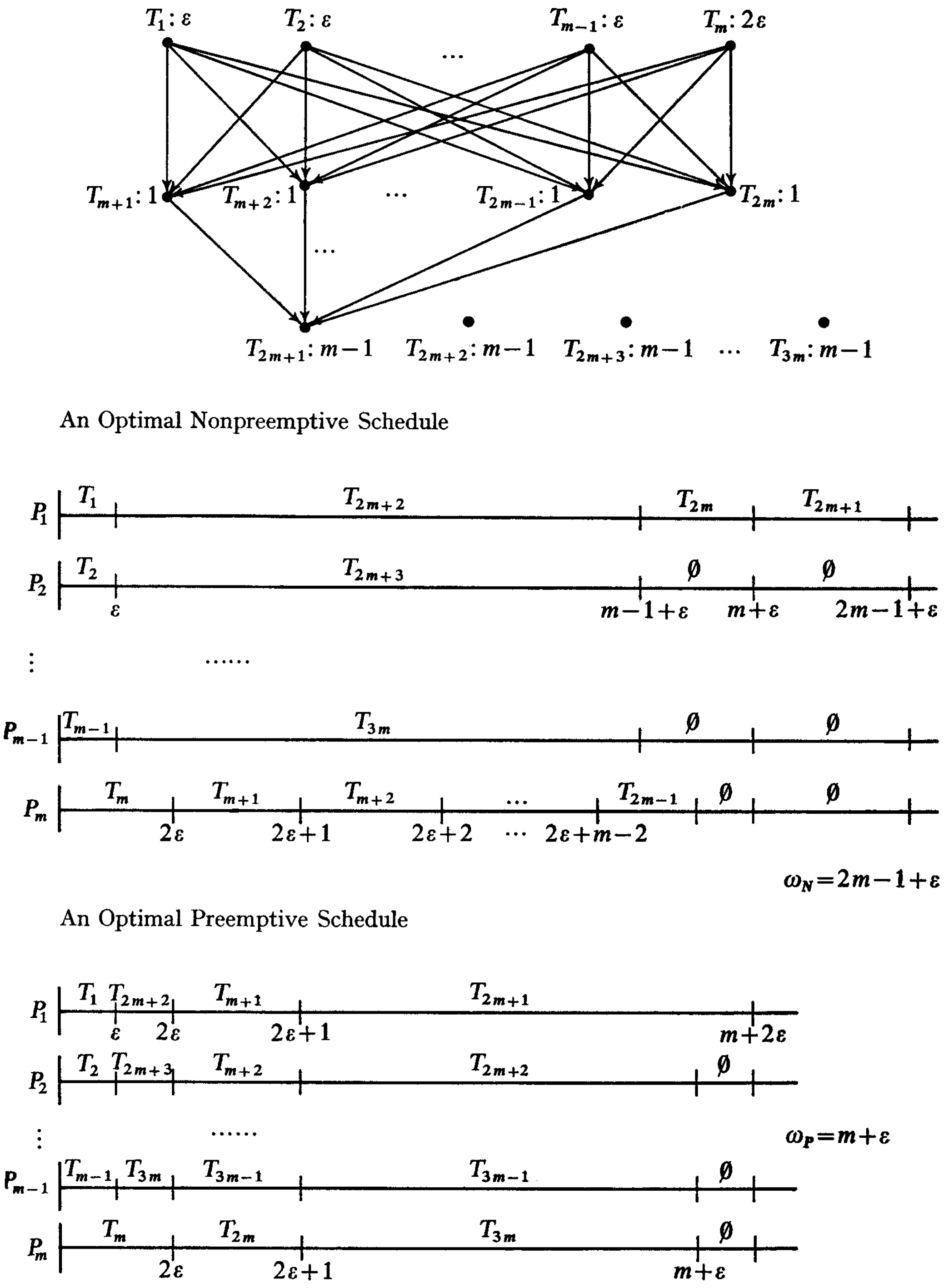

$T_1: \varepsilon$
$T_2: \varepsilon$
$\cdots$
$T_{m-1}: \varepsilon$
$T_m: 2\varepsilon$
$T_{m+1}: 1$
$T_{m+2}: 1$
$\cdots$
$T_{2m-1}: 1$
$T_{2m}: 1$
$\cdots$
$T_{2m+1}: m-1$
$T_{2m+2}: m-1$
$T_{2m+3}: m-1$
$\cdots$
$T_{3m}: m-1$
An Optimal Nonpreemptive Schedule
P_1 T_1 T_{2m+2} T_{2m} T_{2m+1}
P_2 T_2 T_{2m+3} $\emptyset$ $\emptyset$
ε $m-1+\varepsilon$ $m+\varepsilon$ $2m-1+\varepsilon$
$\vdots$ ......
P_{m-1} T_{m-1} T_{3m} $\emptyset$ $\emptyset$
P_m T_m T_{m+1} T_{m+2} $\cdots$ T_{2m-1} $\emptyset$ $\emptyset$
2ε $2\varepsilon+1$ $2\varepsilon+2$ $\cdots$ $2\varepsilon+m-2$
$\omega_N=2m-1+\varepsilon$
An Optimal Preemptive Schedule
P_1 T_1 T_{2m+2} T_{m+1} T_{2m+1}
ε 2ε $2\varepsilon+1$ $m+2\varepsilon$
P_2 T_2 T_{2m+3} T_{m+2} T_{2m+2} $\emptyset$
$\vdots$ ......
$\omega_P=m+\varepsilon$
P_{m-1} T_{m-1} T_{3m} T_{3m-1} T_{3m-1} $\emptyset$
P_m T_m T_{2m} T_{3m} $\emptyset$
2ε $2\varepsilon+1$ $m+\varepsilon$

Fig. 8. Preemptive vs. nonpreemptive schedules

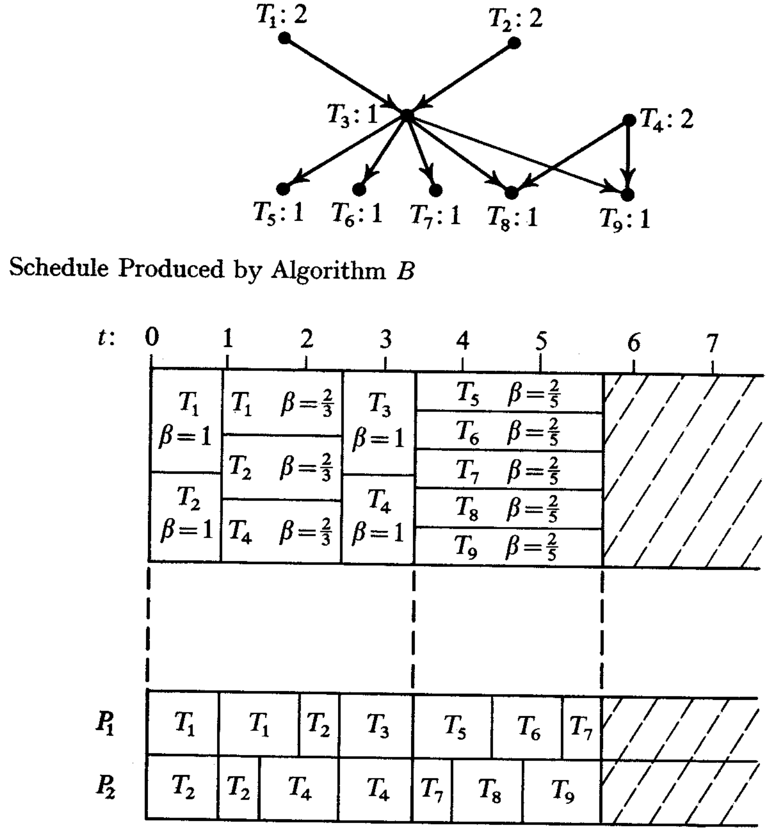
of a longest path from T_i to a terminal task. Now consider the following scheduling algorithm for an arbitrary graph G and m processors.

Algorithm B. Assign one processor each to the tasks at the highest level. If there is a tie among b tasks (because they are at the same level) for the last a ($a < b$) processors then assign a/b of a processor to each of these b tasks. Whenever either of the two events described below occurs, we reassign the processors to the unexecuted portion of the graph G according to the above rule. These are

Event 1. A task is completed.

Event 2. We reach a point where, if we were to continue the present assignment we would be executing some tasks at a lower level at a faster rate than other tasks at a higher level.

An example is shown in Fig. 9 along with the equivalent preemptive schedule. From the figure we can make the following observations. First, at $t = 1$ we have an occurrence of Event 2 (this is the only such occurrence). Second, a conversion


T_1: 2
T_2: 2
T_3: 1
T_4: 2
T_5: 1
T_6: 1
T_7: 1
T_8: 1
T_9: 1
t: 0 1 2 3 4 5 6 7
T_1 $\beta=1$
T_2 $\beta=1$
T_1 $\beta=\frac{2}{3}$
T_2 $\beta=\frac{2}{3}$
T_4 $\beta=\frac{2}{3}$
T_3 $\beta=1$
T_4 $\beta=1$
T_5 $\beta=\frac{2}{5}$
T_6 $\beta=\frac{2}{5}$
T_7 $\beta=\frac{2}{5}$
T_8 $\beta=\frac{2}{5}$
T_9 $\beta=\frac{2}{5}$
P_1 T_1 T_1 T_2 T_3 T_5 T_6 T_7
P_2 T_2 T_2 T_4 T_4 T_7 T_8 T_9

Schedule Produced by Algorithm B

Optimal Preemptive Schedule

Fig. 9. Optimal preemptive schedules

from processor-sharing to a preemptive schedule can be performed individually for each assignment interval over which the processor assignments do not change. Finally, as expected it is never necessary to preempt more frequently than once every $w/2=1/2$ time unit in the resulting preemptive schedule.

Informally, Algorithm B might be termed a generalized "critical path" algorithm. Tasks are given priority based on their distance from terminal tasks. When there are several tasks with the same priority competing for a smaller number of processors, the processors work on all the tasks at a reduced rate which is the same for each task. For $m=2$ Algorithm B defines an optimal preemptive schedule for G. (The algorithm is also optimal for arbitrary m if G is a rooted tree structure.) This is proved elsewhere [4], but the result is easily motiviated by considering the nature of Algorithm A.

First, we observe that Algorithm A at each decision point always sequences the highest-level available tasks first. This follows from the fact that if the task T_i has a higher level than task T_j (we are assuming that all tasks have equal execution times), then $\alpha(T_i)>\alpha(T_j)$. (A simple induction argument suffices to establish this property of the labeling procedure.) Now consider the sequencing produced by Algorithm A for the graph $G_{w/n}$ as $n\to\infty$. Since a given task in G can only retain its scheduling priority over a time interval w/n we note that the ordering of task labels produced by the labeling algorithm *at a given level* has less importance as w/n becomes small. Thus, because of the highest-level-first property it is apparent that in the limit Algorithm A converges to Algorithm B. This gives intuitive support to the fact that the limiting behavior of Algorithm A applied to $G_{w/n}$ does produce an optimal processor-sharing schedule.

References

1. Fulkerson, D. R.: Scheduling in project networks. Proc. IBM Scientific Computing Symposium on Combinatorial Problems. New York: IBM Corporation 1966.
2. Clark, W.: The Gantt chart (3rd Edition). London: Pitman and Sons 1952.
3. Hu, T. C.: Parallel sequencing and assembly line problems. Operations Research **9**, No. 6 (Nov. 1961).
4. Muntz, R. R., Coffman, E. G., Jr.: Optimal preemptive scheduling on two-processor systems. IEEE Trans. on Computers C **18**, No. 11, 1014–1020, Nov. 1969.
5. — Scheduling of computations on multiprocessor systems: The preemptive assignment discipline. PhD. Thesis, Electrical Eng. Dept., Princeton University, April 1969.
6. Graham, R. L.: Bounds for certain multiprocessing anomalies. BSTJ, Nov. 1966, pp. 1563–1581.
7. — Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. **17**, No. 2, 416–429 (1969).
8. Fujii, M., Kasami, T., Ninomiya, K.: Optimal sequence of two equivalent processors. SIAM J. Appl. Math. **17**, No. 3, 784–789 (1969).
9. — Erratum. SIAM J. Appl. Math. **20**, No. 1, 141 (1971).
10. Edmonds, J.: Paths, trees and flowers. Canad. J. Math. **17**, 449–467 (1965).
11. Lawler, E. L. (personal communication).

Ass. Prof. E. G. Coffman, Jr.
Computer Science Department
Pennsylvania State University
University Park, Penn. 16802
U.S.A.

Dr. R. L. Graham
Bell Telephone Laboratories, Inc.
Murray Hill
New Jersey
U.S.A.