Scheduling Independent Tasks To Reduce Mean Finishing Time

J. Bruno, E.G. Coffman Jr., and R. Sethi
The Pennsylvania State University

Sequencing to minimize mean finishing time (or mean time in system) is not only desirable to the user, but it also tends to minimize at each point in time the storage required to hold incomplete tasks. In this paper a deterministic model of independent tasks is introduced and new results are derived which extend and generalize the algorithms known for minimizing mean finishing time. In addition to presenting and analyzing new algorithms it is shown that the most general mean-finishing-time problem for independent tasks is polynomial complete, and hence unlikely to admit of a non-enumerative solution.

Key Words and Phrases: minimizing mean finishing time, minimizing mean flow time, sequencing algorithms, optimal scheduling algorithms, deterministic scheduling models

CR Categories: 4.32, 5.39

I. Introduction

In this paper we study the problem of sequencing independent tasks on $m \geq 1$ processors to minimize the mean finishing time (mean time in system). The importance of the mean-finishing-time criterion is that its minimization in any given schedule tends to reduce the mean number of unfinished tasks at each point in the schedule [1]. Reducing the inventory of unfinished tasks implies in turn that the storage requirements (as a function of time) are reduced.

Of course, it is also desirable to maximize system throughput by minimizing schedule lengths (latest finishing times). For this reason we shall also propose and characterize an algorithm which produces schedules having desirable properties in terms of both the mean-finishing-time and schedule-length criteria. It will be seen that our results will apply directly only to systems such as closed job-shops or those performing real-time control or large and complex numerical computations, for which task execution times are known in advance. However, from our deterministic models some insight into the corresponding problems of other dynamic systems can be gained.

In order to specify the results of subsequent sections in more detail we need the following notation. In the scheduling context of interest we are given a set of $n \geq 1$ independent tasks (or jobs), $\mathfrak{I} = \{T_1, \ldots, T_n\}$, and a set of $m \geq 1$ processors $P_1, \ldots, P_m$. The execution time of task T_j on processor P_i will be denoted by τ_{ij} where τ_{ij} is a nonnegative integer. We use $[\tau_{ij}]$ to denote the $m \times n$ matrix of processing times. We use the familiar Gantt chart, illustrated in Figure 1, to represent schedules. If S is a schedule, $s_j(S)$ and $f_j(S)$ denote the starting time and finishing time, respectively, of task T_j in S. The mean finishing time or mean flow time (mft) of a schedule S is given by $\bar{t}(S) = \sum_{j=1}^{n} f_j(S)$. (It is perhaps more common to find $\bar{t}(S)/n$ as the definition of mean finishing time. Clearly, our simpler and more convenient definition exploits the fact that $\bar{t}(S)/n$ will be minimized whenever $\bar{t}(S)$ is.) The maximum finishing time or length of a schedule S is denoted by $t(S) = \max_j \{f_j(S)\}$. An example for $m = 3$ and $n = 8$ is shown in Figure 1.

In the next section we show that an efficient algorithm exists for finding a schedule S which minimizes $\bar{t}(S)$. Section III considers a more general problem in which the tasks are assumed to have different priorities or urgency measures. In particular, let w_j be a weight associated with T_j which measures the relative urgency of finishing T_j. The problem now becomes one of sequencing $\mathfrak{I}$ on $m \geq 1$ processors so that we minimize the weighted mean finishing time (wmft) $\bar{t}(S) = \sum_{j=1}^{n} w_j f_j(S)$.

We show in Section III that this problem is in the well-known class of "polynomial complete" problems, and hence a non-enumerative solution is not likely to be found.

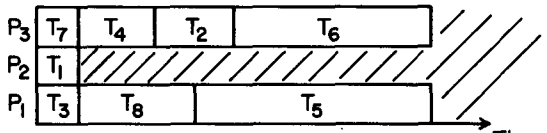
In Section IV, we characterize the lengths of schedules on identical processors for which $\bar{t}(S)$ is minimum. The results presented motivate Section V, in which an algorithm is proposed for producing schedules that have favorable mean- and maximum-finishing-time properties. Bounds on the performance of the algorithm are derived and the results of a simulation are described.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

Authors' address: Department of Computer Science, The Pennsylvania State University, University Park, PA 16802.

Fig. 1. An example for $m = 3, n = 8$.

$$[\tau_{ij}] = \begin{bmatrix} 2 & 3 & 1 & 4 & 6 & 5 & 2 & 3 \\ 1 & 4 & 1 & 5 & 5 & 4 & 3 & 4 \\ 3 & 2 & 3 & 2 & 3 & 5 & 1 & 2 \end{bmatrix}$$



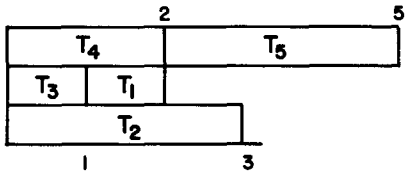
SCHEDULE S

$f_1(S) = 1$ $f_5(S) = 10$
$f_2(S) = 5$ $f_6(S) = 10$
$f_3(S) = 1$ $f_7(S) = 1$
$f_4(S) = 3$ $f_8(S) = 4$

$\bar{t}(\hat{S}) = 35$

Fig. 2. Example of an optimal schedule.

$$[\tau_{ij}] = \begin{bmatrix} 2 & 3 & 1 & 4 & 6 \\ 1 & 4 & 1 & 5 & 5 \\ 3 & 2 & 3 & 2 & 3 \end{bmatrix} \quad Q = \begin{bmatrix} 2 & \textcircled{3} & 1 & 4 & 6 \\ \textcircled{1} & 4 & 1 & 5 & 5 \\ 3 & 2 & 3 & 2 & \textcircled{3} \\ 4 & 6 & 2 & 8 & 12 \\ 2 & 8 & \textcircled{2} & 10 & 10 \\ 6 & 4 & 6 & \textcircled{4} & 6 \\ 6 & 9 & 3 & 12 & 18 \\ 3 & 12 & 3 & 15 & 15 \\ 9 & 6 & 9 & 6 & 9 \\ 8 & 12 & 4 & 16 & 24 \\ 4 & 16 & 4 & 20 & 20 \\ 12 & 8 & 12 & 8 & 12 \\ 10 & 15 & 5 & 20 & 30 \\ 5 & 20 & 5 & 25 & 25 \\ 15 & 10 & 15 & 10 & 15 \end{bmatrix}$$



$\bar{t}(\hat{S}) = 13$

II. Reduction to a Transportation Problem

Our objective is to obtain an efficient (non-enumerative) algorithm which takes $[\tau_{ij}]$ as input and constructs a schedule $\hat{S}$ such that $\bar{t}(\hat{S})$ is as small as possible. Let S be some schedule and suppose task T_j is run on machine P_i and that there are exactly $k - 1$ jobs which are scheduled after T_j on P_i. We can write $\bar{t}(S)$ in such a way as to isolate all those terms which depend on τ_{ij}. It is not difficult to see that the coefficient of τ_{ij} is k; that is, $\bar{t}(S)$ is the sum of $k\tau_{ij}$ and $n - 1$ other terms not containing τ_{ij}. If T_j runs last on P_i, then the coefficient of τ_{ij} is 1; if T_j runs next-to-last on P_i then the coefficient of τ_{ij} is 2, etc. This observation leads to the following related problem.

From the $m \times n$ matrix $[\tau_{ij}]$ of processing times we form the matrix

$$Q = \begin{bmatrix} [\tau_{ij}] \\ 2[\tau_{ij}] \\ \vdots \\ n[\tau_{ij}] \end{bmatrix}$$

where $p[\tau_{ij}]$ denotes the matrix obtained from $[\tau_{ij}]$ by multiplying each element of $[\tau_{ij}]$ by p. The matrix Q is $nm \times n$. Elements of Q are denoted by q_{ij} where $1 \leq i \leq nm$ and $1 \leq j \leq n$. Intuitively, the elements in $[\tau_{ij}]$ represent the possible contributions to the mean finishing time from jobs scheduled last on their respective machines, $2[\tau_{ij}]$ represents the possible contributions to the mean finishing time from jobs scheduled next-to-last on their respective machines, etc.

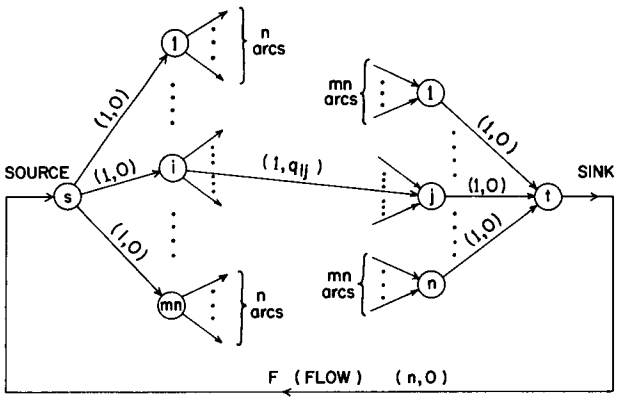
A set of n elements $q_{i_1 1}, \ldots, q_{i_n n}$ is called a *feasible set* if no two elements are in the same row; the *cost* of such a set is given by $\sum_{r=1}^{n} q_{i_r r}$. The problem is to find a feasible set with smallest possible cost. For we will show that if $q_{i_1 * 1}, \ldots, q_{i_n * n}$ is a feasible set with smallest cost then this set determines a schedule S^* such that $\bar{t}(S^*)$ is minimal.

Without loss of generality, we can assume that if $i_r^* > m$, then there exists an $s (1 \leq s \leq n)$ such that $i_s^* = i_r^* - m$. To see this, suppose on the contrary that $i_r^* > m$ and no element from row $i_r^* - m$ is in the feasible set. Since $q_{i_r^* r} > q_{i_r^* - m, r}$ and we can therefore replace $q_{i_r^* r}$ by $q_{i_r^* - m, r}$ in the feasible set without increasing the cost, we arrive at a contradiction. (In the degenerate case when entries in a column are zero, we assume i_r^* is as small as possible.)

Note that the numbers i_r^* can be written uniquely as $i_r^* = (k_r - 1)m + \Delta_r$ where $1 \leq k_r \leq n$ and $1 \leq \Delta_r \leq m$. The numbers k_r and Δ_r determine the position of the T_r in an optimal schedule; T_r is the k_rth task from the last to be run on machine Δ_r. An example is given in Figure 2. An optimal schedule $\hat{S}$ is determined by the circled elements of Q according to the definitions just given for k_r and Δ_r. This schedule, which has a cost 13, is also shown in the figure.

The problem of finding a minimal cost feasible set for a matrix Q can be reformulated as a transportation problem. The network is shown in Figure 3. The arcs are labeled with pairs (x,y) where x is the arc capacity and y is the cost per unit flow. It is easy to see that $F_{\max}$, the maximum flow, is equal to n. The problem here is to find a minimal cost flow pattern which achieves the maximum total flow ($F_{\max}$) through the network. It is only necessary to consider maximum flow patterns which saturate all the arcs having a nonzero flow. In this case it is easy to see that a minimal-cost-flow pattern which achieves the maximum flow determines a minimal-cost-feasible set for the matrix Q. An element q_{ij} is in the feasible set if and only if the arc from vertex i to

Fig. 3. A transportation network.



vertex j is saturated. This problem is a special case of the Hitchcock problem [2]. Thus, an efficient (non-enumerative) algorithm for finding an optimal schedule $\hat{S}$ can be constructed from the known techniques for solving the Hitchcock problem [3].

III. The Weighted-Finishing-Time Criterion

When demand on system resources is high, competing tasks are often assigned on the basis of their priorities. Thus system routines tend to receive preferential treatment, as well as users willing to pay more for better service. As indicated in Section I, we can model this situation by assigning priority or urgency measures w_j to each task T_j. The problem now becomes the minimization of the *wmft*, $\sum_{j=1}^{n} w_j f_j(S)$. We will show that this model leads to a "polynomial complete problem" for the case of identical processors. We let τ_j be the execution time of T_j on each of the P_i $(1 \leq i \leq m)$.

Reducibility between combinatorial problems has been the subject of considerable attention in the recent past. We refer the reader to [4, 5, 6] for a definition of the term "polynomial complete" and problems that are known to be polynomial complete. The value of demonstrating that a problem is complete is that the existence of a non-enumerative solution for the problem would imply the existence of non-enumerative solutions for a significantly large class of problems which include, for example, the traveling salesman problem, the knapsack problem, bin-packing problems, and finding the chromatic number of a graph. Thus, there is impressive circumstantial evidence that it will be difficult to find non-enumerative solutions for any polynomial complete problems.

In order to show that our *wmft* problem is polynomial complete, we use the fact that the following problem is known to be polynomial complete [5].

Knapsack Problem. Given the set of positive integers $B = \{a_1, a_2, \ldots, a_n\}$ and an integer b, does there exist a subset $C = \{c_1, c_2, \ldots, c_j\}$ of B, such that $\sum_{i=1}^{j} c_i = b$? If C exists the knapsack problem will be said to have a solution. □

Note that the knapsack problem has a yes-no answer. In general, polynomial complete problems are posed so that a yes-no answer is called for. We are led to the following definition of our scheduling problem.

Problem P1. We have two (identical) processors, tasks $T_1, \ldots, T_n$, execution times $\tau_1, \ldots, \tau_n$, and weights $w_1, \ldots, w_n$. For a given number k, to be called the *limit*, we ask: Does there exist a schedule S such that $\sum_{i=1}^{n} w_i f_i(S) \leq k$? □

We will show that the knapsack problem can be couched in the framework of problem P1. Clearly, an efficient solution of P1 would then imply an efficient solution of the knapsack problem. Given an instance of the knapsack problem, we can define the following problem.

Problem P2. Given a set of integers $\{a_1, \ldots, a_n\}$ and the integer b, let $A = \sum_{i=1}^{n} a_i$. We are given tasks $T_0, T_1, \ldots, T_n, T_{n+1}$, execution times

$$\tau_0 = 2b - 1$$
$$\tau_i = 2a_i \quad \text{for } i = 1, \ldots, n$$
$$\tau_{n+1} = 2A - 2b - 1$$

and weights $w_0, \ldots, w_{n+1}$ given by $2b - 1$, $2a_1$, $\ldots$, $2a_n$, $2A - 2b - 1$. We have two identical processors and the *limit* $(\sum_{i=0}^{n+1} \sum_{j=0}^{i} w_i w_j) - (2A - 1)^2$. Does there exist a schedule S such that the *wmft* of S is no greater than the above limit? □

Since there are two identical processors, the $n + 2$ tasks will be partitioned into two sets, one for each processor. The following result will simplify our view of schedules produced for problem P2.

PROPOSITION 1. *The wmft of a schedule S for problem P2 is independent of the order of execution of tasks assigned to a given processor.*

This result follows from the fact that $\tau_i/w_i = 1$ for $i = 0, \ldots, n + 1$, and therefore the interchange of two adjacent tasks does not alter the *wmft* [1]. It remains to observe that any permutation can be realized by a sequence of interchanges.

Proposition 1 assures us that the *wmft* of a schedule for problem P2 depends only on the way tasks are partitioned into the set for P_1 and the set for P_2. Consequently, a *schedule* may be viewed as a *pair* (S_1, S_2), where $S_1 \cup S_2 = \{T_0, T_1, \ldots, T_{n+1}\}$ and $S_1 \cap S_2 = \varnothing$. Informally, the tasks in S_1 are executed on P_1, and the tasks in S_2 on P_2. Let $M(S_1, S_2)$ denote the *wmft* of schedule (S_1, S_2).

It will be convenient for us to extend the notion of weights and execution times to sets of tasks. Thus, if $S = \{T_{i_1}, T_{i_2}, \ldots, T_{i_j}\}$ is a set of tasks then $w(S) = \sum_{k=1}^{j} w_{i_k}$ and $\tau(S) = \sum_{k=1}^{j} \tau_{i_k}$. We now need a mechanism for determining the *wmft* for a schedule.

PROPOSITION 2. *Let* (S_1, S_2) *and* (S_1', S_2') *be two schedules for problem P2 such that* $S_2 \subseteq S_2'$. *Then* $M(S_1', S_2') = M(S_1, S_2) - w(S_1 - S_1')[\tau(S_1') - \tau(S_2)]$.

PROOF. First note that $S_1 \cap S_2 = S_1' \cap S_2' = \varnothing$. Since $S_2 \subseteq S_2'$, it follows that $S_2' - S_2 = S_1 - S_1'$. From Proposition 1, we can assume without loss of generality that tasks in the set $S_1 - S_1'$ are executed

last on P_1. Consider a task T, with execution time τ and finishing time f, in $S_1 - S_1'$. The finishing time f will be given by $\tau(S_1')$ plus the finishing times of all tasks preceding and including T, in $S_1 - S_1'$. Thus, the contribution to $M(S_1, S_2)$ by tasks in $S_1 - S_1'$ is given by $\tau(S_1')\cdot w(S_1 - S_1') + k$, where k depends only on the set $S_1 - S_1'$.

Now k will be $M(S_1 - S_1', \varnothing)$, or the *wmft* of a schedule that deals only with tasks in $S_1 - S_1'$, and assigns them all to P_1. Now assume without loss of generality that tasks in $(S_1 - S_1')$ are executed last in (S_1', S_2') on P_2. The contribution of tasks in $S_1 - S_1'$ to $M(S_1', S_2')$ will be $\tau(S_2)\cdot w(S_1 - S_1') + k$. The proof now follows directly. □

Proposition 2 provides us with a means of calculating the *wmft* of a schedule for problem P2. It is easy to compute $M(S, \varnothing)$, the case when all tasks are assigned to the same processor. Using $M(S, \varnothing)$, we can invoke Proposition 2 to compute $M(S_1, S_2)$, for any schedule (S_1, S_2). Since the execution times of all tasks are the same as the weights, we get $M(S_1, S_2) = M(S, \varnothing) - \tau(S_2)\cdot\tau(S_1)$. Since $\tau(S_1) + \tau(S_2) = 4A-2$, we let $\tau(S_1) = 2A + k - 1$. Hence, $\tau(S_2) = 2A - k - 1$ and $\tau(S_1)\cdot\tau(S_2) = (2A-1)^2 - k^2$. It is easy to see that $M(S, \varnothing) = \sum_{i=0}^{n+1}\sum_{j=0}^{i} w_i w_j$, and thus $M(S_1, S_2) = \sum_{i=0}^{n+1}\sum_{j=0}^{i} w_i w_j - (2A-1)^2 + k^2$. It follows that the limit of problem P2 can be achieved if and only if $k = 0$; i.e. $\tau(S_1) = \tau(S_2) = 2A - 1$. Alternatively, we have from the above discussion the following proposition.

PROPOSITION 3. *The limit of problem P2 can be achieved if and only if the maximum finishing time of a schedule for the set of tasks is* $2A - 2$.

Note that the sum of the execution times of all tasks in problem P2 is $4A - 2$. Thus the maximum finishing time for any schedule is at least $2A - 1$ and we have the following.

PROPOSITION 4. *A schedule S with maximum-finishing-time* $2A - 1$ *exists for problem P2 if and only if the knapsack problem has a solution.*

PROOF. Assuming the knapsack problem has a solution, we will show that a maximum finishing time of of $2A - 1$ is achievable. Since we are assuming that the knapsack problem has a solution, there must be some subset $\mathfrak{I}'$ of the tasks, such that $\tau(\mathfrak{I}') = 2b$. Construct a schedule S as follows: Assign task T_{n+1} and the tasks in $\mathfrak{I}'$ to P_1. Since $\tau_{n+1} = 2A - 2b - 1$, we have $\tau(\mathfrak{I}') + \tau_{n+1} = 2A - 1$. Assigning all remaining tasks to P_2, we get the maximum finishing time, $2A - 1$.

Now suppose that there exists a schedule with maximum-finishing-time $2A - 1$. We will show that the knapsack problem has a solution. Consider tasks T_0 and T_{n+1} and suppose they are assigned to the same processor. Since $\tau_0 = 2b - 1$ and $\tau_{n+1} = 2A - 2b - 1$, we would need to assign another task of execution time 1 unit to the same processor in order to make the maximum-finishing-time $2A - 1$. But, all other tasks have execution times of at least 2 units. Therefore T_0 and T_{n+1} must be assigned to different processors. Since $\tau_{n+1} = 2A - 2b - 1$, in order to achieve a maximum finishing time of $2A - 1$, the other tasks assigned to the same processor as T_{n+1} must have execution times totaling 2b. Therefore the knapsack problem has a solution. □

Propositions 3 and 4 demonstrate that the knapsack problem "reduces" to problem P1, in the sense of [5]. It is now easy to show that P1 is indeed polynomial complete.

As a corollary of proposition P4, we can infer that another scheduling problem of interest—that of determining the maximum finishing time of a schedule for a set of independent tasks on two processors—is polynomial complete. One of the notable facets of the mean-finishing-time problem in the previous section is that it admits a non-enumerative solution. If instead of mean finishing time, the parameter of interest is the maximum finishing time; then we have seen that the problem becomes polynomial complete. For some other polynomial complete scheduling problems, see [7].

IV. Maximum-Finishing-Time Comparisons of SPT and LPT

In this section we restrict ourselves to identical processors and use the notation of the preceding section. It is well known [1] that the shortest-processing-time (SPT) scheduling discipline defined below minimizes the mean finishing time for $m \geq 1$ processors.

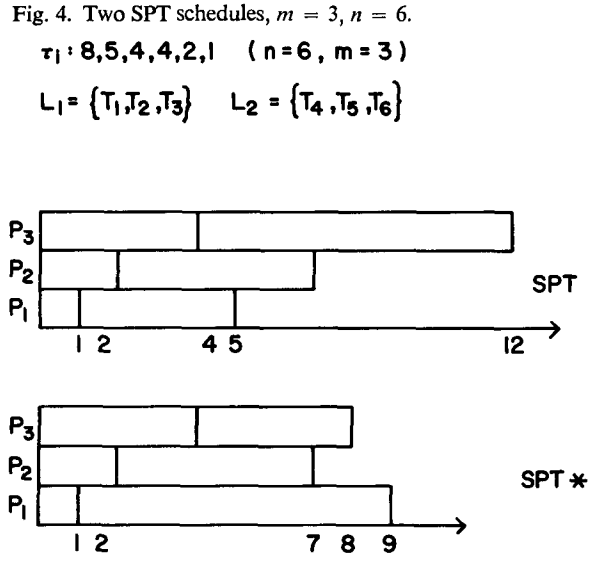
Suppose the tasks are ordered such that $\tau_1 \geq \tau_2 \geq \cdots \geq \tau_n$, and let n be a multiple of m (if it is not, we can always add tasks of zero length to the collection). Define the sets L_r for $r = 1, \ldots, n/m$ as

$$L_1 = \{T_1, \ldots, T_m\}$$
$$L_2 = \{T_{m+1}, \ldots, T_{2m}\}$$
$$\vdots$$
$$L_{n/m} = \{T_{n-m+1}, \ldots, T_n\}$$

The SPT discipline schedules the tasks in L_r all on different processors (there are $m!$ ways to do this) and such that these tasks are processed rth from the last on their respective processors. If S is an SPT schedule, then we are guaranteed that S minimizes the mean finishing time. Among all the schedules obtainable according to the SPT discipline some are better than others with respect to the maximum finishing time. An example is shown in Figure 4 with two SPT schedules. For a given set of tasks $\mathfrak{I}$, we shall let SPT* denote an SPT schedule which minimizes the maximum finishing time among all SPT schedules.

The antithetical rule, largest-processing-time (LPT) scheduling, produces schedules which tend to maximize mean finishing time but minimize the maximum finishing time. A schedule S is an LPT schedule if for every T_i and T_j in S, $\tau_i \geq \tau_j$ implies $s_i(S) \leq s_j(S)$. We next present bounds on the relative effectiveness of SPT, LPT, and optimal scheduling (OPT).

Fig. 4. Two SPT schedules, $m = 3$, $n = 6$.



PROPOSITION 5. *For a given set of tasks* $\mathfrak{I} = \{T_1, \ldots, T_n\}$ *and* $m \geq 1$, *let OPT be a schedule which gives the smallest maximum finishing time among all schedules. Then*

$$1 \leq t(\text{SPT}^*)/t(\text{OPT}) \leq 2 - 1/m. \tag{1}$$

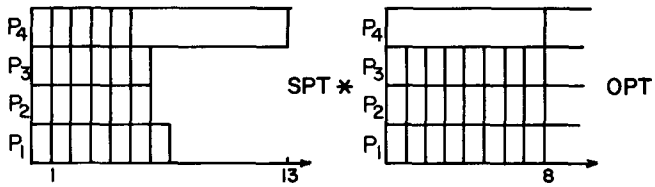
Moreover, these bounds are best possible.

The lower bound in (1) is obvious. Any collection of equal-length tasks shows that it is a best bound. The upper bound follows from a result on list-scheduling anomalies [8] which states that the ratio of maximum finishing times for any two list schedules is upper-bounded by $2 - 1/m$ (however, this is not generally a best bound). The fact that (1) is a best bound for any given m is shown by the set $\mathfrak{I}_k$ ($k \geq 1$) of tasks for which $n = m(m-1)k + 1$ and

$$\tau_1 = mk,$$
$$\tau_i = 1, \quad i = 2, \ldots, n.$$

Figure 5 shows the example for $m = 4$ and $k = 2$. In general the ratio for this class of examples is $t(\text{SPT}^*)/t(\text{OPT}) = 2 - 1/m - 1/km$, which approaches the bound as k gets large.

Fig. 5. Example in proof of proposition 5.



It is interesting to find that the same upper bound applies when LPT scheduling is considered instead of optimal scheduling, and that the bound is still best possible. The proof of this fact is essentially the same as before.

PROPOSITION 6. *Let* $\mathfrak{I} = \{T_1, \ldots, T_n\}$ *be a given set of tasks and* $m \geq 1$. *Then*

$$t(\text{SPT}^*)/t(\text{LPT}) \leq 2 - 1/m, \tag{2}$$

and this bound is the best possible. It is not true, however, that $t(\text{LPT}) \geq t(\text{SPT}^*)$ for all $\mathfrak{I}$; that is, there are collections of tasks such that $t(\text{SPT}^*) < t(\text{LPT})$. It has been shown that [10] $t(\text{SPT}^*)/t(\text{LPT}) \geq (4m - 1)(5m - 2)$. That the ratio $(4m - 1)/(5m - 2)$ can be achieved for all m is shown by an example having $n = 2m + 1$ tasks where

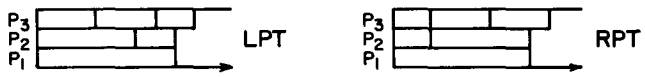
$$\tau_i = 3m - 1 - i, \quad i = 1, \ldots, m,$$
$$\tau_i = 3m - i, \quad i = m + 1, \ldots, 2m,$$
$$\tau_n = m.$$

In comparing LPT with optimal scheduling Graham [9] proved the following bound on the maximum-finishing-time performance of LPT: $1 \leq t(\text{LPT})/t(\text{OPT}) \leq (4m - 1)/3m$, $m \geq 1$. In view of earlier results this supports the general superiority of LPT over SPT sequencing in producing short schedules. In the next section, we introduce an algorithm with the above maximum-finishing-time characteristic but with a mean-finishing-time characteristic that is considerably better than LPT.

V. The RPT Algorithm

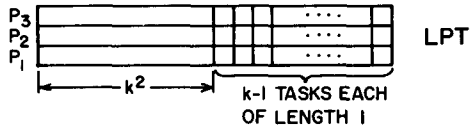
A natural approach to obtaining an algorithm having desirable properties with respect to both maximum and mean finishing time is to consider the left-justified, reverse-LPT (to be denoted RPT) schedule. The principal motivation is that, although it is not generally SPT, it is SPT with respect to the tasks assigned to the individual processors. Thus, RPT is a scheduling algorithm with the maximum-finishing-time properties of LPT which one might expect to have mean-finishing-time performance approaching that of SPT. An example RPT schedule is shown in Figure 6. In the following, we characterize the mean-finishing-time performance of RPT.

Fig. 6. An example of an RPT schedule.



Since tasks are in SPT order on each processor in an RPT schedule, the optimality of SPT implies $\bar{t}(\text{SPT}) \leq \bar{t}(\text{RPT}) \leq \bar{t}(\text{LPT})$ for any set of tasks and a given $m \geq 1$. Moreover, for a given m, the ratio $\bar{t}(\text{LPT})/\bar{t}(\text{RPT})$ can be made arbitrarily large. This is illustrated in Figure 7. As can be readily verified the ratio $\bar{t}(\text{LPT})/\bar{t}(\text{RPT})$ in-

Fig. 7. $\bar{t}$(LPT)/$\bar{t}$(RPT) increases as k increases.



creases with increasing k. To compare $\bar{t}$(RPT) and $\bar{t}$(SPT), we have the following result, whose proof can be found in the Appendix.

PROPOSITION 7. *For any set of tasks* $\mathfrak{J}$ *and a given* $m \geq 1$, *we have*

$$1 \leq \bar{t}(\text{RPT})/\bar{t}(\text{SPT}) \leq m. \tag{3}$$

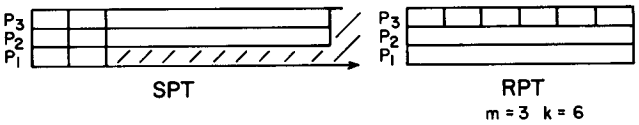
Moreover, these bounds are the best possible.

The example in Figure 8 shows that the bound is best possible. The execution times are

$$\tau_i = 1, \quad 1 \leq i \leq k,$$
$$\tau_i = k, \quad k+1 \leq i \leq m+k-1,$$

where k is assumed to be a multiple of m. It is readily verified that $\bar{t}_R/\bar{t}_S = m(2(m-1)+(k+1))/(2(m^2-1)+(k+m))$ which, by our choice of k, can be made to approach m arbitrarily closely. It is somewhat surprising that $\bar{t}$(RPT)/$\bar{t}$(SPT) can be as much as m.

Fig. 8. Worst case example for $\bar{t}$(RPT)/$\bar{t}$(SPT).



The proof of this fact employs a very special "job mix" (see Figure 8), which may not be representative of job mixes encountered in practice. Experimental results show that, in spite of the theoretical bound, the mean finishing time of RPT normally compares very favorably to SPT. The results of our experimentation are described in Table I. Note especially that even with the large-variance geometric distribution, we consistently obtain values of $\bar{t}_{\text{RPT}}$ within about 2 percent of the optimal.

Table I.

Moments of the mean finishing-time ratio distributions with 1000 jobs, whose run times, j, are distributed with geometric probability mass function, $p(1-p)^{j-1}$.

m	n	p	Mean of $\bar{t}_{\text{SPT}}/\bar{t}_{\text{RPT}}$	Standard deviation $\bar{t}_{\text{SPT}}/\bar{t}_{\text{RPT}}$
2	12	1/8	0.9885	0.02751
4	24	1/8	0.9851	0.02382
6	36	1/8	0.9843	0.01956
4	24	1/16	0.9805	0.02759
4	24	1/8	0.9851	0.02382
4	24	1/4	0.9857	0.02380
4	32	1/8	0.9924	0.01464
6	32	1/8	0.9814	0.02187

Acknowledgment. We are pleased to acknowledge the work of Arthur Lenskold, who is responsible for Table I.

Appendix. Proof of Proposition 7

The lower bound in (3) is assured by the optimality of SPT; any collection of equal-length tasks shows that (3) is a best bound. The upper bound in (3) will be shown by induction on the number, n, of tasks being scheduled. The upper bound obviously holds for $n = 1, 2, \ldots, m+1$. Suppose it holds for all task sets whose size is less than n, and let $\mathfrak{J}$ be a set of n tasks. Let $\Sigma_R(i)$ be the sum of task completion times in an RPT schedule for $\mathfrak{J}(i) = \mathfrak{J} - \{T_1, T_2, \ldots, T_{n-i}\}$, where $\mathfrak{J} \equiv \mathfrak{J}(n)$ and the tasks are assumed to be indexed so that $\tau_1 \leq \tau_2 \leq \cdots \leq \tau_n$. Define $\Sigma_S(i)$ similarly for an SPT schedule.

It is not difficult to verify that $\Sigma_S(n) = \Sigma_S(n-1) + h_S(n)\tau_1$ and $\Sigma_R(n) = \Sigma_R(n-1) + h_R(n)\tau_1$ where $h_S(n)$ ($h_R(n)$) denotes the total number of tasks with nonzero processing time assigned to the processor on which T_1 is executed in the SPT (RPT) schedule. Thus, the mean-flow-time ratio for $\mathfrak{J}$ can be written

$$\begin{aligned}\eta_n \Sigma_R(n)/\Sigma_S(n) &= (\Sigma_R(n-1) \\ &\quad + h_R(n)\tau_1)/(\Sigma_S(n-1) + h_S(n)\tau_1) \\ &= (\eta_{n-1} + \alpha h_R(n))/(1 + \alpha h_S(n))\end{aligned} \tag{6}$$

where $\alpha = \tau_1/\Sigma_S(n-1)$.

Obviously, $h_R(n) \leq n$ for all $m \leq 1$. Since it is possible to choose S such that $h_S(n) = \lfloor \frac{n}{m} \rfloor$, we have $h_R(n) \leq m\, h_S(n)$. Using this inequality in (6) we get $\eta_n \leq (\eta_{n-1} + m\, h_S(n))/(1 + h_S(n))$. From the induction hypothesis $\eta_{n-1} \leq m$, and hence $\eta_n \leq (m + m\, h_S(n))/(1 + h_S(n)) = m$, thus completing the proof that m is a bound. The example discussed following Proposition 7 shows that m can be approached arbitrarily closely, and therefore m is a best bound. □

References

1. Conway, R.W., Maxwell, W.L., and Miller, L.W. *Theory of Scheduling*. Addison-Wesley, New York, 1900.
2. Ford, L.R., and Fulkerson, D.R. *Flows in Networks*. Princeton U. Press, Princeton, N.J., 1962.
3. Bruno, J. A scheduling algorithm for minimizing mean flow time. Tech. Rep. #141, Comput. Sci. Dept., Penn State U., University Park, Penna., July 1973.
4. Cook, S.A. The complexity of theorem proving procedures. 3rd ACM Symp. on Theory of Computing, Shaker Heights, Oh., May 1971, pp. 151–158.
5. Karp, R.M. Reducibility between combinatorial problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (Eds) Plenum Press, New York 1972, pp. 85–103.
6. Sahni, S. Some related problems from network flows, game theory and integer programming. 13th Ann. Symp. on Switching and Automata Theory, Oct. 1972, pp. 130–139.
7. Ullman, J.D. Polynomial complete scheduling problems TR9. Dept. of Comput. Sci., U. of California at Berkeley, Mar. 1973.
8. Graham, R.L. Bounds for certain multiprocessing anomalies. *Bell Syst. Tech J.* (Nov. 1966), 1563–1581.
9. Graham, R.L. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math 17*, 2 (Mar. 1969), 416–429.
10. Coffman, E.G. On a conjecture concerning the comparison of SPT and LPT scheduling. Tech. Rep. #140, Comput. Sci. Dept., Penn State U., University Park, Penna., July 1973.