Bounds on multiprocessing anomalies and related packing algorithms

by R. L. GRAHAM

Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

INTRODUCTION

It has been known for some time that certain rather general models of multiprocessing systems frequently exhibit behavior which could be termed "anomalous," e.g., an *increase* in the number of processors of the system can cause an *increase* in the time used to complete a job.[42,36,20] In order to fully realize the potential benefits afforded by parallel processing, it becomes important to understand the underlying causes of this behavior and the extent to which the resulting system performance may be degraded.

In this paper we survey a number of theoretical results obtained during the past few years in connection with this topic. We also discuss many of the algorithms designed either to optimize or at least to improve the performance of the multiprocessor system under consideration.

The performance of a system or an algorithm can be measured in several rather different ways.[5] Two of the most common involve examining the *expected* behavior and the *worst-case* behavior of the object under consideration. Although knowledge of expected behavior is generally more useful in typical day-to-day applications, theoretical results in this direction require assumptions concerning the underlying probability distributions of the parameters involved and historically have been extremely resistant to attack.

On the other hand, there are many situations for which worst-case behavior is the appropriate measure (in addition to the fact that worst-case behavior does bound expected behavior). This type of analysis is currently a very active area of research, and theoretical insight into the worst-case behavior of a number of algorithms from various disciplines is now beginning to emerge (e.g., see References 7, 14, 15, 19, 20, 21, 26, 27, 29, 32, 33, 44, 47, and especially Reference 41.) It is this latter measure of performance which will be used on the models and algorithms of this paper. Since it is essential to have on hand the worst examples one can think of before conjecturing and (hopefully) proving bounds on worst-case behavior, numerous such examples will be given throughout the text.

Before concluding this section, it seems appropriate to make a few remarks concerning the general area of these topics. Recent years have seen the emergence of a vital and important new discipline, often called "analysis of algorithms." As its broad objective, it seeks to obtain a deeper understanding of the nature of algorithms. These investigations range, for example, from the detailed analysis of the behavior of a specific sorting routine, on one hand, to the recent negative solution† to Hilbert's Tenth Problem by Matijasevič[37] and Julia Robinson,[43] on the other. It is within this general framework that the present discussion should be viewed.

A GENERAL MULTIPROCESSING SYSTEM

Let us suppose we have n (abstract) identical processors P_i, $i=1, \ldots, n$, and we are given a set of tasks $\mathfrak{I}=\{T_1, \ldots, T_r\}$ which is to be processed by the P_i. We are also given a partial order†† $<$ on $\mathfrak{I}$ and a function $\mu:\mathfrak{I}\to(0, \infty)$. Once a processor P_i begins to execute a task T_j, it works without interruption until the completion of that task,††† requiring altogether $\mu(T_j)$ units of time. It is also required that the partial order be respected in the following sense: If $T_i<T_j$ then T_j cannot be started until T_i has been completed. Finally, we are given a sequence $L=(T_{i_1}, \ldots, T_{i_r})$, called the

† Which roughly speaking shows that there is no universal algorithm for deciding whether a diophantine equation has solutions or not.

†† See Reference 31 for terminology.

††† This is known as *nonpreemptive* scheduling as opposed to *preemptive* scheduling in which the execution of a task may be interrupted.[5]

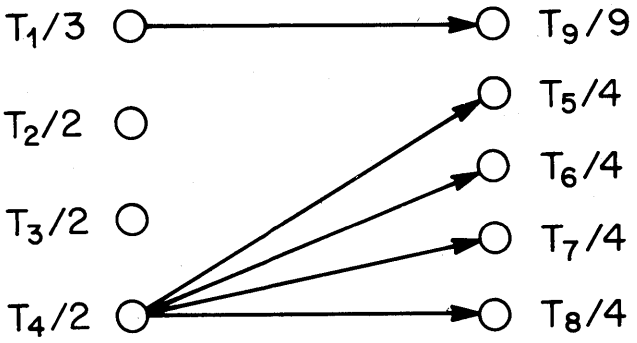

Figure 1—A simple graph G

(*priority*) *list* (or schedule) consisting of some permutation of all the tasks $\mathfrak{z}$. The P_i execute the T_j as follows: Initially, at time 0, all the processors (instantaneously) scan the list L from the beginning, searching for tasks T_i which are "ready" to be executed, i.e., which have no predecessors under $<$. The first ready task T_j in L which P_i finds immediately begins to be executed by P_i; P_i continues to execute T_j for the $\mu(T_j)$ units of time required to complete T_j. In general, at any time a processor P_i completes a task, it immediately scans L for the first available ready task to execute. If there are currently no such tasks, then P_i becomes idle. We shall also say in this case that P_i is executing an *empty task* which we denote by φ (or φ_i). P_i remains idle until some other P_k completes a task, at which time P_i (and, of course, P_k) immediately scans L for ready tasks which may now exist because of the completion of T_j. If two (or more) processors both attempt to start executing a task, it will be our convention to assign the task to the processor with the smaller index. The least time at which all tasks of T have been completed will be denoted by ω.

We consider an example which illustrates the working of the preceding multiprocessing system and various anomalies associated with it. We indicate the partial order $<$ on T and the function μ by a *directed graph*†

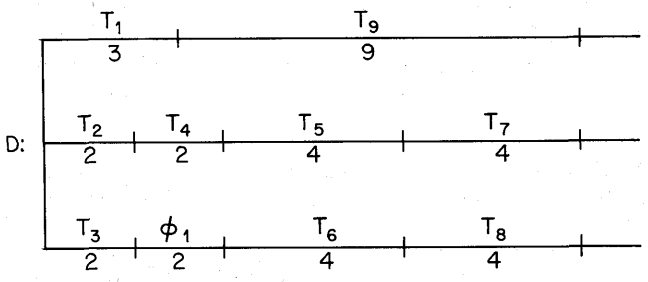

Figure 2—The timing diagram D for G

† For terminology in graph theory, see Reference 23.

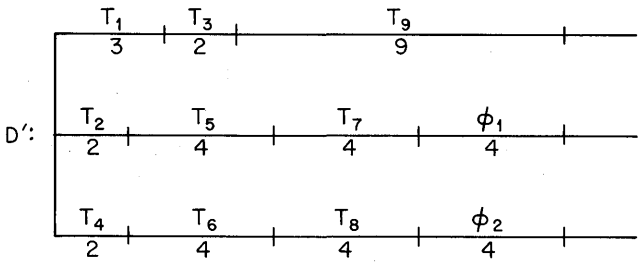

Figure 3—The timing diagram D′ when L′ is used

$G(<, \mu)$. In $G(<, \mu)$ the vertices correspond to the T_i and a directed edge from T_i to T_j denotes $T_i < T_j$. The vertex T_j of $G(<, \mu)$ will usually be labeled with the symbols $T_j/\mu(T_j)$. The activity of each P_i is conveniently represented by a *timing diagram* D (also known as a Gantt chart.[2] D consists of n horizontal half-lines (labeled by the P_i) in which each line is a time axis starting from time 0 and is subdivided into segments labeled according to the corresponding activity of the P_i. In Figure 1 we show a simple graph G.

In Figure 2, we give the corresponding timing diagram D assuming that the list $L = (T_1, T_2, \ldots, T_9)$ is used with three processors. The finishing time is $\omega = 12$.

Note that on D we have labeled the intervals above by the task and below by the length of time needed to execute the task.

It is evident from the definition of ω that it is a function of L, μ, $<$ and n. Let us vary each of these four parameters in the example and see the resulting effect this variation has on ω.

(i) Replace L by $L' = (T_1, T_2, T_4, T_5, T_6, T_3, T_9, T_7, T_8)$, leaving μ, $<$ and n unchanged (Figure 3). For the new list L', $\omega' = \omega'(L', \mu, <, n) = 14$.

(ii) Change $<$ to $<'$ by removing $T_4 < T_5$ and $T_4 < T_6$.

For the new partial order $<'$, $\omega' = \omega'(L, \mu, <', n) = 16$ (Figure 4).

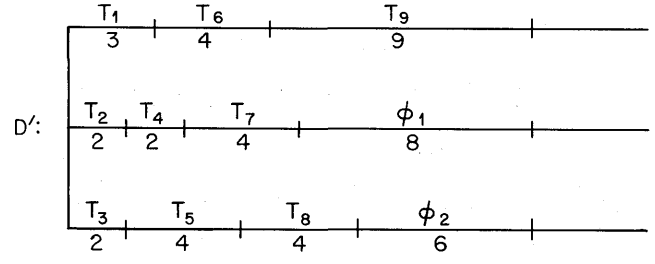

Figure 4—The timing diagram D′ when $<'$ is used

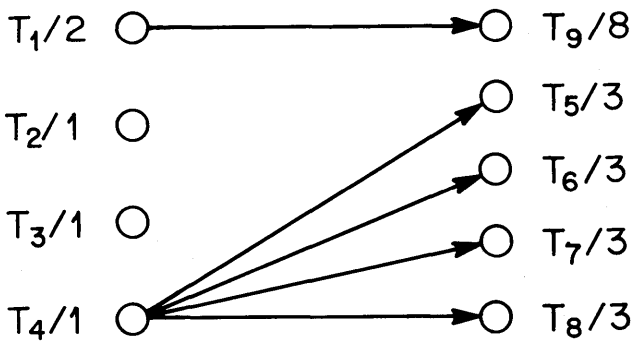

Figure 5—The new graph G using μ'

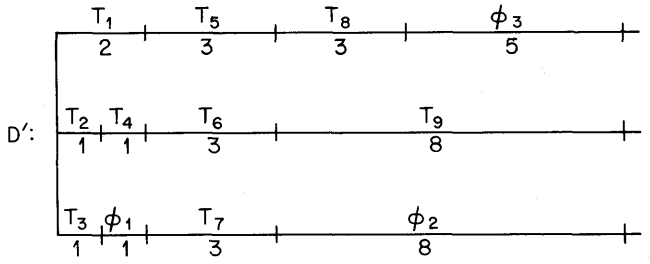

Figure 6—The timing diagram D′ when μ' is used

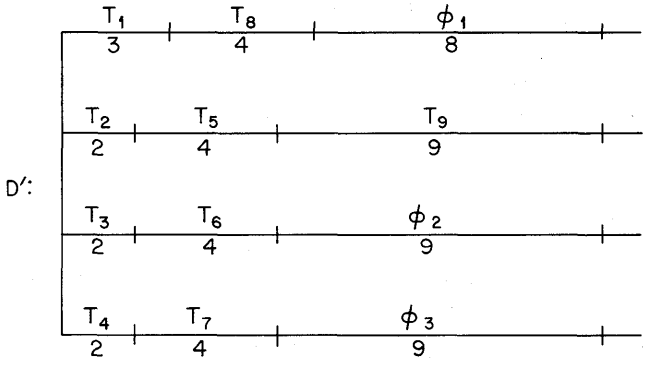

Figure 7—The timing diagram D′ when 4 processors are used

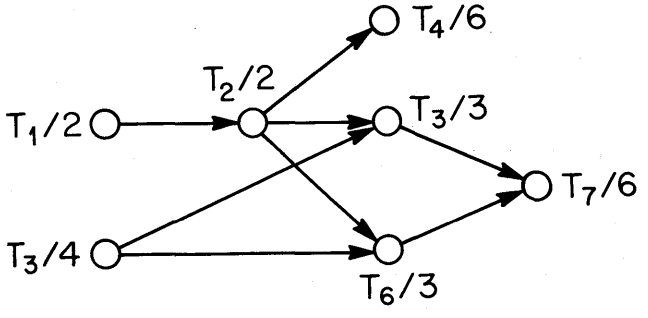

Figure 8—Another simple graph

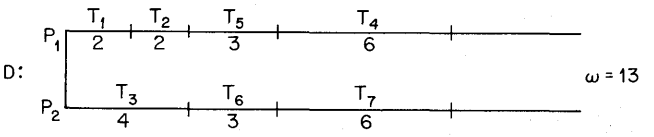

Figure 9—The timing diagram D using the list L

(iii) Decrease μ to μ' by defining $\mu'(T_i) = \mu(T_i) - 1$ for all i. In this case $G(<, \mu)$ is shown in Figure 5.

The corresponding timing diagram D' is shown in Figure 6 where we see $\omega' = \omega'(L, \mu', <, n) = 13$.

(iv) Increase n from three to four (Figure 7). In this case $\omega' = 15$!

Note that in (iii) by using the list $L'' = (T_1, T_2, T_3, T_4, T_9, T_5, T_6, T_7, T_8)$ we can reduce the finishing time to 10 which is less than the 12 of the original unshortened problem. This does not always have to occur though, as the example given in Figure 8 shows.

When the (optimal) list $L = (T_1, T_2, T_3, T_5, T_6, T_4, T_7)$ is used, $\omega = 13$ (Figure 9).

If all execution times are decreased by one unit then *all* lists generate the same finishing time $\omega' = 14$. A typical D' is shown in Figure 10.

A GENERAL BOUND

The examples in the preceding section show that contrary to what might generally be expected, *relaxing* $<$, *decreasing* μ or *increasing* n can all cause ω to *increase*. It is natural to inquire into the extent to which these changes can affect the finishing time ω. The right measure turns out to be the ratio of the possible finishing times, and a bound is given in the following result.

Theorem.[19,20]

Suppose we are given a set of tasks $\mathfrak{I}$, which we wish to execute twice. The first time we use a time function μ, a partial order $<$, a list L and a multiprocessing system composed of n processors. The second time we

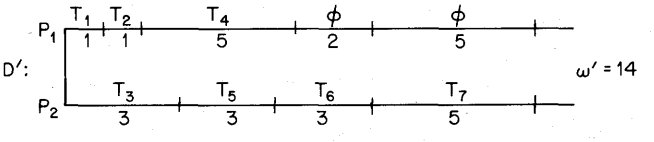

Figure 10—A timing diagram D using shortened times

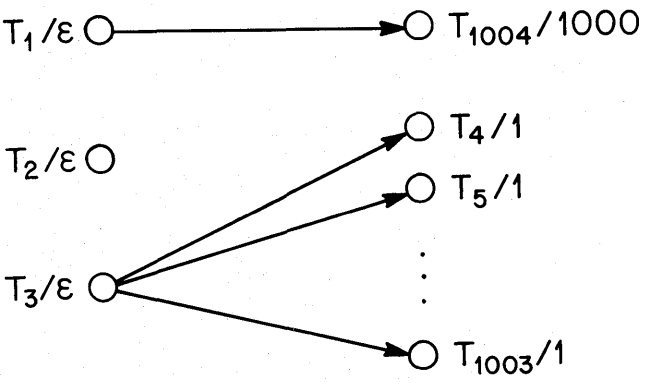

Figure 11—A graph G

use a time function $\mu' \leq \mu$, a partial order† $<' \subseteq <$, a list L' and a multiprocessing system composed of n' processors. Let ω and ω' denote the respective finishing times. Then

$$\omega'/\omega \leq 1+(n-)1/n'. \qquad (1)$$

Furthermore, this bound is best possible in the sense that the right-hand side cannot be replaced by any smaller function of n and n'.

For $n=n'$, the bound becomes

$$\omega'/\omega \leq 2-1/n \qquad (2)$$

In fact, examples†† can be given[19] which show that the bound in (2) can be achieved (to within an arbitrary $\epsilon>0$) by varying any *one* of the three parameters L, μ and $<$. Note that (2) implies that using the worst possible list instead of the best possible list still only results in an increase in ω of at most a factor of $2-1/n$.

When $n=1$, (1) implies $\omega'/\omega \leq 1$ which agrees with the obvious fact that the aforementioned changes in L, μ, $<$ and n can never cause increase in the running time when compared to that for single processor. On the other hand, when $n>1$ then even a large increase

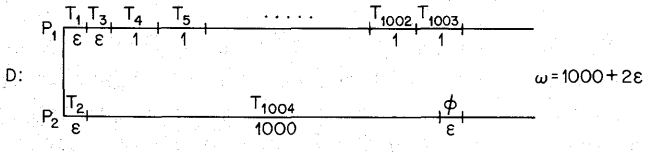

Figure 12—2 processors are used to execute the tasks of G

in the number of processors (but with a fixed list L) can cause ω to increase as the example given in Figure 11 shows (where $0<\epsilon<1$).

If the tasks of G are executed by two processors using the list $L=(T_1, T_2, T_3, \ldots, T_{1004})$ then $\omega=1000+2\epsilon$ (Figure 12).

If the tasks of G are executed by 1000 processors using the same list L then $\omega'=1001+\epsilon$ (Figure 13). However, it is always true that the use of a suitable list will prevent ω from increasing because of an increase in n (e.g., in this case, take $L=(T_{1004}, T_1, T_2, \ldots, T_{1003})$).

We mention here that even if inserted idleness and preemption† are allowed in the first run, it can be shown that ω/ω' is still bounded above by $2-1/n$.[3,18]

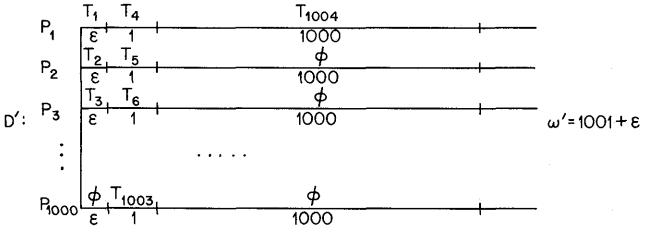

Figure 13—1000 processors are used to execute the tasks of G

SOME SPECIAL BOUNDS

We next consider results arising from attempts at finding lists which keep the ratio of ω/ω_0 close to one. Since for any given problem, there are just finitely many possible lists then one might be tempted to say: "Examine them all and select the optimum." Not much insight is needed, however, to realize that due to the explosive growth of functions such as $n!$, this is not a realistic solution. What one is looking for instead is an algorithm which will guarantee that ω/ω_0 is reasonably close to one provided we are willing to expend an appropriate amount of energy applying the algorithm. Unfortunately, no general results of this type presently exist. There is a special case, however, in which steps in this direction have been taken. This is the case in which $<$ is empty, i.e., there are no precedence constraints between the tasks. We shall restrict ourselves to this case for the remainder of this section.

Suppose, for some (arbitrary) k, the k tasks with the largest values of μ have been selected† and somehow

† Since a partial order on $\mathfrak{I}$ is a subset of $\mathfrak{I} \times \mathfrak{I}$, $<' \subseteq <$ has the obvious meaning.

†† Recent examples of M. T. Kaufman (personal communication) show that in many cases the bounds of (1) and (2) can be achieved even when $\mu(T) = 1$ for all T.

† i.e., holding a processor idle when it could be busy and interrupting the execution of a task before completion.

† Recent results of Blum, Floyd, Pratt, Rivest and Tarjan[41] allow this to be done in no more than $6r$ binary comparisons where r denotes the number of tasks.

arranged in a list L_k which is optimal with respect to this set of k tasks (i.e., for no other list can this set of k tasks finish earlier). Form the list $L^{(k)}$ by adjoining the remaining tasks arbitrarily but so that they follow all the tasks of L_k. Let $\omega(k)$ denote the finishing time using this list with a system of n processors. If ω_0 denotes the global optimum, i.e., the minimum possible finishing time over all possible lists then the following result holds.

Theorem.[20]

$$\frac{\omega(k)}{\omega_0} \leq 1+\frac{1-1/n}{1+[k/n]} \tag{3}$$

For $k \equiv 0 \pmod{n}$ this bound is best possible.

For the case of $k \equiv 0 \pmod{n}$ the following example establishes the optimality of the bound from below.

For $1 \leq i \leq k+1+n(n-1)$, define $\mu(T_i)$ by

$$\mu(T_i) = \begin{cases} n & \text{for } 1 \leq i \leq k+1, \\ 1 & \text{for } k+2 \leq i \leq k+1+n(n-1). \end{cases}$$

For this set of tasks and the list $L(k) = (T_1, \ldots, T_k, T_{k+2}, \ldots, T_{k+1+n(n-1)}, T_{k+1})$ we have $\omega(k) = k+2n-1$.

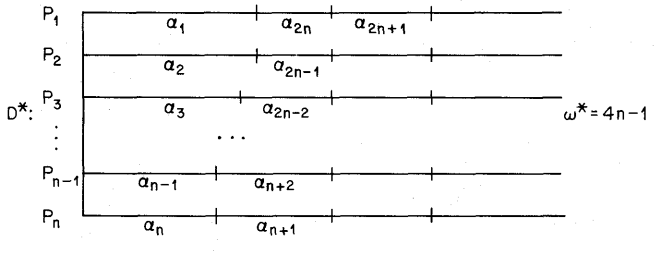

Figure 14—The timing diagram D* using the decreasing list L*

Since $\omega_0 = k+n$, and $k \equiv 0 \pmod{n}$ then

$$\frac{\omega(k)}{\omega_0} = 1+\frac{1-1/n}{1+[k/n]}$$

as asserted.

For $k=0$, (3) reduces to (2) while for $k=n$ we have

$$\omega(n)/\omega_0 \leq 3/2-1/2n. \tag{4}$$

The required optimal assignment of the largest n tasks to the n processors is immediate—just assign each of these tasks to a different processor. For $k=2n$, (3) reduces to

$$\omega(2n)/\omega_0 \leq 4/3-1/3n, \tag{5}$$

a bound which will soon be encountered again.

An important property of (3) is that the right-hand side tends to 1 as k gets larger compared to n. Thus, in order for $\omega(k)$ to be assured of being within 10 percent of the minimum value ω_0, for example, it suffices to optimally schedule the largest $9n$ tasks.

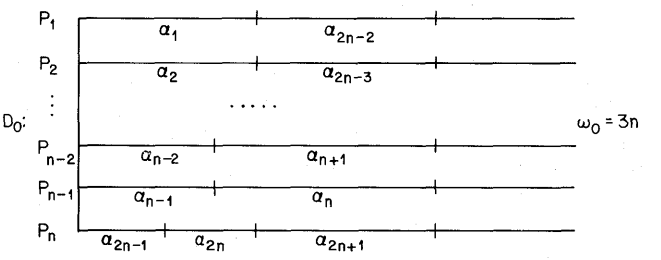

Figure 15—The timing diagram D_0 using an optimal list

Another heuristic technique for approximating the optimal finishing time ω_0 is to use the "decreasing" list† $L^* = (T_{i_1}, T_{i_2}, \ldots)$ where $\mu(T_{i_1}) \geq \mu(T_{i_2}) \geq \ldots$. The corresponding finishing time ω^* satisfies the following inequality.

Theorem.[20]

$$\omega^*/\omega_0 \leq 4/3-1/3n. \tag{6}$$

This bound is best possible.

For $n=2$, (6) yields a bound of 7/6 which is exactly the ratio obtained from the canonical example with five tasks having execution times of 3, 3, 2, 2 and 2. More generally, the following example shows that (6) is exact. $\mathfrak{I}$ consists of $r=2n+1$ independent tasks T_k with $\mu(T_k) = \alpha_k = 2n-[(k+1)/2]$ for $1 \leq k \leq 2n$ and $\mu(T_{2n+1}) = \alpha_{2n+1} = n$ (where $[x]$ denotes the greatest integer not exceeding x). Thus

$$(\alpha_1, \alpha_2, \ldots, \alpha_{2n+1})$$
$$= (2n-1, 2n-1, 2n-2, 2n-2, \ldots, n+1, n+1, n, n, n)$$

In Figure 14, $\mathfrak{I}$ is executed using the decreasing list $L^* = (T_1, T_2, \ldots, T_{2n+1})$.

In Figure 15, $\mathfrak{I}$ is executed using an optimal list.

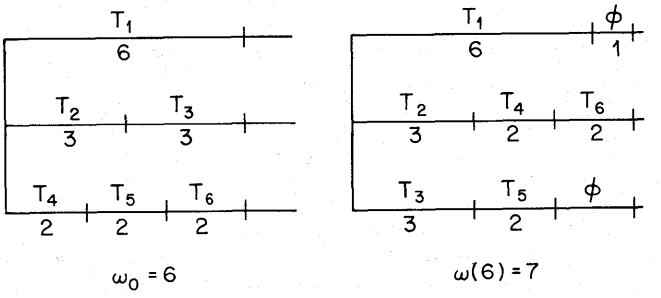

Figure 16—Example illustrating difference between L* and L(2n)

† Such a list can be formed in essentially no more than $r \log r/\log 2$ binary comparisons.[33]

It is interesting to observe that although the right-hand sides of (5) and (6) are the same, arranging the tasks by decreasing execution time does not necessarily guarantee that the largest $2n$ tasks will be executed optimally by L^*. An example showing this (due to J. H. Spencer (personal communication)) is given in Figure 16. The execution times of the tasks are (6, 3, 3, 2, 2, 2) and three processors are used.

The following result shows that if none of the execution times is large compared to the sum of all the execution times then ω^* cannot be too far from ω_0.

Theorem.[18]

If $<$ is empty and

$$\max \mu(T) / \sum_T \mu(T) \leq \beta$$

then

$$\omega^*/\omega_0 \leq 1+n\beta. \tag{7}$$

Another approach is to start with a timing diagram resulting from some arbitrary list and then make *pairwise interchanges* between tasks executed by pairs of processors which decrease the finishing time, until this can no longer be done. If ω' denotes the final finishing time resulting from this operation then it can be shown[18] that

$$\omega'/\omega_0 \leq 2-2/(n+1) \tag{8}$$

and, furthermore, this bound cannot be improved.

SOME ALGORITHMS FOR OPTIMAL LISTS

There seems little doubt that even for the case when $<$ is empty, $\mu(T)$ is an integer, and $n=2$, any algorithm which determines an optimal list for any set of tasks must be essentially enumerative† in its computational efficiency. This problem can be rephrased as follows:

Given a sequence $S=(s_1, \ldots, s_r)$ of positive integers find a choice of $\epsilon_i = \pm 1$ such that

$$\left| \sum_{k=1}^{r} \epsilon_k s_k \right|$$

is minimized.

Thus any hope for efficient algorithms which produce optimal schedules for more general multiprocessing problems seems remote indeed. There are several special cases, however, for which such algorithms exist.

For the case when $\mu(T)=1$ for all tasks T and $<$ is a forest,† Hu[28] has shown that the following algorithm generates an optimal list L_0.

(i) Define the level $\lambda(T)$ of any terminal†† task T to be 1.
(ii) If T' is an immediate successor of T, define $\lambda(T)$ to be $\lambda(T')+1$.
(iii) Form the list $L_0=(T_{i_1}, T_{i_2}, \ldots, T_{i_r})$ in order of decreasing λ values, i.e., $\lambda(T_{i_1}) \geq \lambda(T_{i_2}) \geq \cdots \geq \lambda(T_{i_r})$.

Theorem.[28]

L_0 is an optimal list when $\mu(T)=1$ for all T and $<$ is a tree.

The only other case for which an efficient algorithm is currently known is when $\mu(T)=1$ for all T, $n=2$ and $<$ is arbitrary. In fact, two quite distinct algorithms have been given. One of these, due to Fujii, Kasami and Ninomiya[11,12] is based on a matching algorithm for bipartite graphs of Edmonds[8] and appears to be of order $O(r^3)$. The other, due to Coffman and Graham,[3] uses the following labeling technique:

Assuming there are r tasks, each task T will be assigned a unique label $\alpha(T) \epsilon \{1, 2, \ldots, r\}$.

(i) Choose an arbitrary terminal task T_0 and define $\alpha(T_0)=1$.
(ii) Assume the values 1, 2, ..., $k-1$ have been assigned for some $k \leq r$. For each unlabeled task T having all its immediate successors already labeled, form the decreasing sequence $M(T)=(m_1, m_2, \ldots, m_5)$ of the labels of T's immediate successors. These $M(T)$ are lexicographically† ordered. Choose a *minimal* $M(T')$ in this order and define $\alpha(T')=k$.
(iii) Form the list $L^*=(T_{i_1}, T_{i_2}, \ldots, T_{i_r})$ according to decreasing α values, i.e., $\alpha(T_{i_1}) > \alpha(T_{i_2}) > \cdots > \alpha(T_{i_r})$.

Theorem.[3]

L^* is an optimal list when $\mu(T)=1$ for all T and $n=2$.

† More precisely, the number of steps it may require cannot be bounded by a fixed polynomial in the number of bits of information needed to specify the input data.

† This means that every task T has at most one immediate successor T', i.e., $T < T'$ and for no T'' is $T < T'' < T'$. Actually, by adding a dummy task T_0 preceded by all other tasks, $<$ can be made into a *tree*, without loss of generality.

†† i.e., a task with no successor.

† i.e., dictionary order, so that (5, 4, 3) precedes (6, 2) and (5, 4, 3, 2).

This algorithm has been shown to be of order $O(r^2)$ and so, in a sense, is best possible since the partial order $<$ can also have this order of number of elements.

The increase in complexity of this algorithm over Hu's algorithm seems to be due to the greatly increased structure an *arbitrary* partial order may have when compared to that of a tree. Even relatively simple partial orders can defeat many other algorithms which might be thought to be optimal for this case. The example in Figure 17 illustrates this.

An optimal list for this example is $L^* = (T_{10}, T_9, \ldots, T_1)$ where we assume $\mu(T_i) = 1$ for all i. Any algorithm which allows T_{10} not to be executed first is not optimal, as, for example, executing tasks on the basis of the longest chain to a terminal task (i.e., according to levels), or executing tasks on the basis of the largest number of successors.

For $n > 2$ and $\mu(T) = 1$ for all T, the algorithm no longer produces optimal lists as the example in Figure 18 shows.

It would be interesting to know the worst-case behavior of the lists produced by this algorithm for general n.

The current state of affairs here seems to be similar to that of the job-shop scheduling problem[5,30] for which optimal schedules can be efficiently generated when $n = 2$, while for $n > 2$ no satisfactory algorithms are known.

A DUAL PROBLEM

Up to this point, we have generally regarded the number of processors as fixed and asked for the list

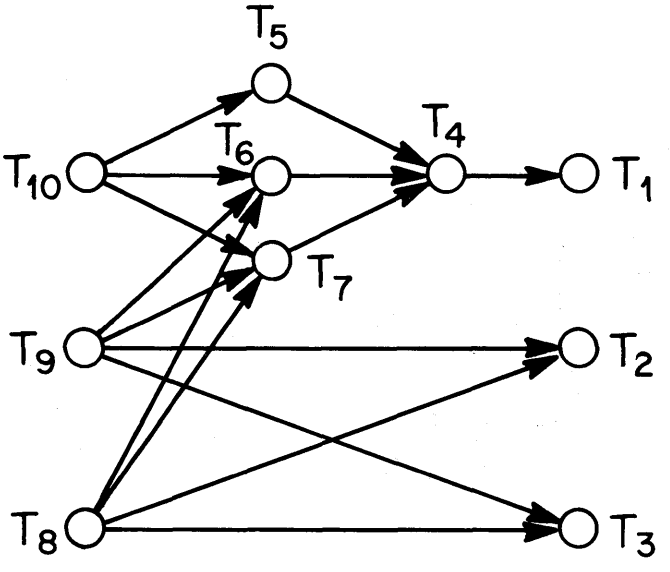

Figure 17—A useful graph for counterexamples

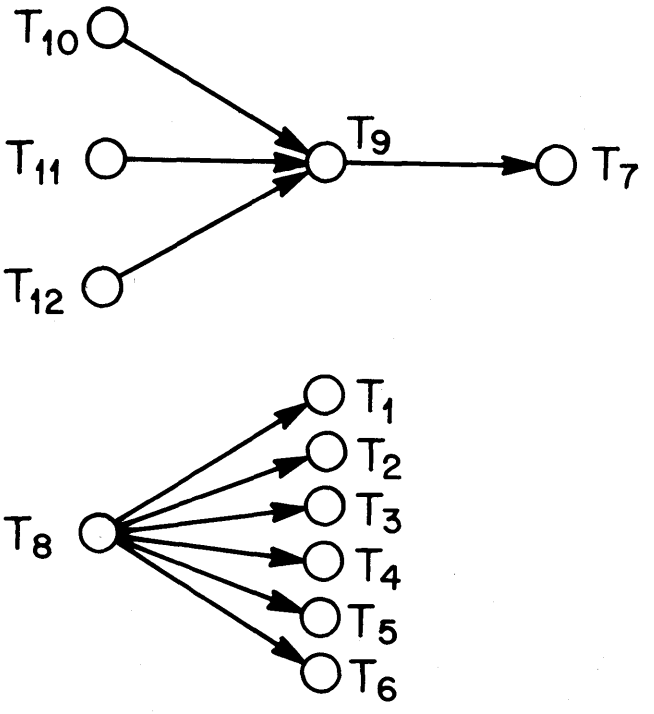

Figure 18—A counterexample to optimality when $n = 3$

L which (approximately) minimizes the finishing time ω. We now invert this question as follows: For a fixed deadline ω^*, we ask for a list L which when used will (approximately) minimize the number of processors needed to execute all tasks by the time ω^*. Of course, the two questions are essentially equivalent, and so no efficient algorithms are known for the general case.† Nevertheless, a number of recent results are now available for several special cases and these will be the subject of this section.

We first make a few remarks concerning the general case. It is not hard to see that if λ^* denotes the length of the longest chain†† in the partially-ordered set of tasks $\mathfrak{I}$, then we must have $\omega^* \geq \lambda^*$. Otherwise, no number of processors could execute all tasks of $\mathfrak{I}$ in ω^* time units. On the other hand, if sufficiently many processors are available then all tasks of $\mathfrak{I}$ can be executed in time λ^*. In fact, if m^* denotes the maximal number of mutually *incomparable*† tasks of $\mathfrak{I}$, then it is never necessary to have more than m^* processors in the system since clearly no more than m^* can be in operation at any one time.

† Both of these questions are raised in Reference 10.

†† i.e., a sequence $T_{i_1} < T_{i_2} < \cdots < T_{i_m}$ with $\sum_k \mu(T_{i_k})$ maximal.

† T_i and T_j are incomparable if neither $T_i < T_j$ nor $T_j < T_i$ hold.

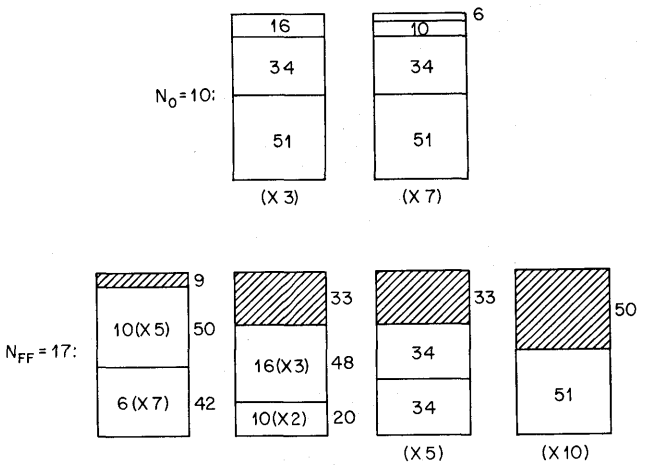

Figure 19—An example with $N_{FF}/N_0 = 17/10$

For the case in which $\mu(T)=1$ for all tasks T, lower bounds for both problems are provided by results of Hu.[28] In this case, if $\lambda(T)$ denotes the *level* of a task T as defined in the preceding section, let m denote the *maximum* level of any task in $\mathfrak{I}$ and for $0\leq k\leq m$, let $\Lambda(k)$ denote the number of tasks having level strictly greater than k.

Theorem.[28]

If n processors can execute $\mathfrak{I}$ with finishing time ω then

$$\omega \geq \max_{0\leq k\leq m} (k+\Lambda(k)/n). \tag{9}$$

For other early results dealing with these and related problems, the reader may consult References 1, 5, 13, 24, 38, 42, 45, and 46.

For the remainder of this section, we restrict ourselves to the special case in which there are no precedence constraints on the tasks. In this case the second problem becomes a special case of the one-dimensional cutting stock problem[15,17] as well as a special case of the assembly-line balancing problem.[5]

We can also think of the problem in the following terms. We are given a set of objects $_i$,O with O_i having weight $\alpha_i=\mu(T_i)$, $1\leq i\leq r$. We have at our disposal an unlimited supply of boxes B_j, each with a maximum capacity of ω^* units of weight. It is required to assign all the objects to the minimum number N_0 of boxes subject to the constraint that the total weight assigned to any box can be no more than ω^*. In this form, this question takes the form of a typical loading or packing problem.[9] Integer linear programming algorithms have been given[9,16,17,25] for obtaining optimal solutions for this problem, but the amount of computation necessary soon becomes excessive as the size of the problem grows.

Several heuristic algorithms have been suggested[9,15,40] for approximating N_0. One of these, which we call the "first-fit" algorithm, is defined as follows: For a given list $L=(\alpha_{i_2}, \alpha_{i_2}, \ldots, \alpha_{i_r})$, the α_{ik} are successively assigned in order of increasing k, each to the box B_j of lowest index into which it can validly be placed. The number of boxes thus required will be denoted by $N_{FF}(L)$, or just N_{FF}, when the dependence on L is suppressed.

If L is chosen so that $\alpha_{i_1}\geq\alpha_{i_2}\geq\cdots\geq\alpha_{i_r}$ then the first-fit algorithm using this list is called the "first-fit decreasing" algorithm and the corresponding $N_{FF}(L)$ is denoted by N_{FFD}.

Instead of first-fit, one might instead assign the next α_k in a list L to the box in which the resulting unused capacity is minimal. This is called the "best-fit" algorithm and N_{BF} will denote the number of boxes required in this case. The corresponding definitions of "best-fit decreasing" and N_{BFD} are analogous to first-fit decreasing and N_{FFD} and are omitted.

One of the first questions which arises concerning these algorithms is the extent by which they can ever deviate from N_0. Only for N_{FF} is the behavior accurately known.

Theorem.[47,15]

For any $\epsilon>0$, if N_0 is sufficiently large then

$$N_{FF}/N_0<17/10+\epsilon. \tag{10}$$

The 17/10 in (10) is best possible. An example for which $N_{FF}/N_0=17/10$ is given in Figure 19 (where

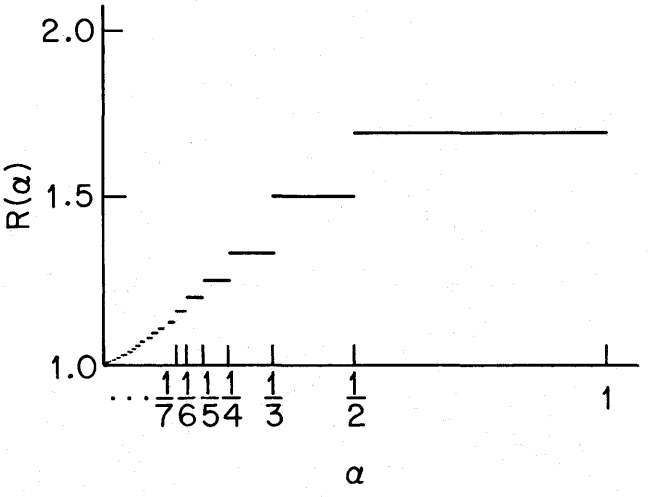

Figure 20—The function R(α)

$\omega^*=101$). The multiplicity of types of boxes and objects are given in parentheses.

For any $\epsilon>0$ examples can be given with $N_{FF}/N_0 \geq 17/10-\epsilon$ and N_0 arbitrarily large. It appears, however, that for N_0 sufficiently large, N_{FF}/N_0 is strictly less than 17/10.

In order to achieve a ratio N_{FF}/N_0 close to 17/10 it is necessary[15] to have some of the α_i exceed $\omega^*/2$. Conversely, if all α_i are small compared to ω^* then N_{FF}/N_0 must be relatively close to one. This is stated precisely in the following result.

Theorem.[15]

Suppose $\max \alpha_i/\omega^* \leq \alpha$. Then for any $\epsilon>0$, if N_0 is sufficiently large then

$$N_{FF}/N_0-\epsilon \leq \begin{cases} 17/10 \text{ for } \alpha > \frac{1}{2}, \\ 1+\lfloor \alpha^{-1} \rfloor^{-1} \text{ for } 0<\alpha \leq \frac{1}{2}. \end{cases} \tag{11}$$

The right-hand side of (11) cannot be replaced by any smaller function of α.

We denote the right-hand side of (11) by $R(\alpha)$ and illustrate it in Figure 20.

It is conjectured[15] that the worst-case behavior of N_{BF}/N_0 is the same as that of N_{FF}/N_0 but this has not yet been established. It is known that $R(\alpha)$ is also a lower bound for N_{BF}/N_0 when $\max \alpha_i \leq \alpha\omega^*$.

As one might suspect, N_{FFD}/N_0 cannot differ from 1 by as much as N_{FF}/N_0 can. This is shown in the following result.

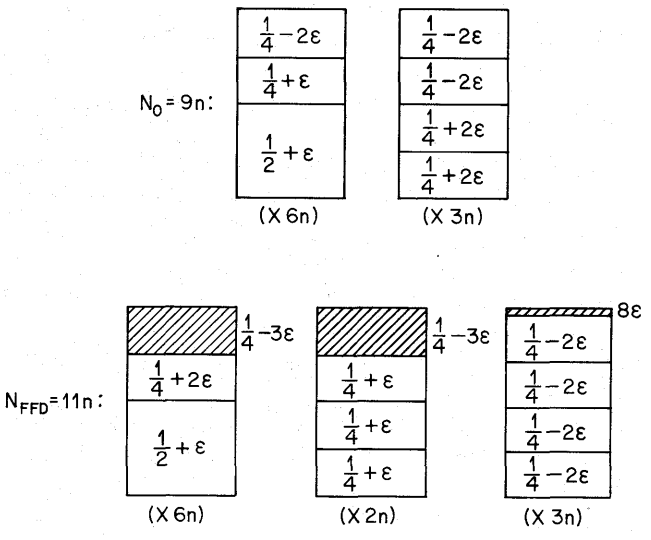

Figure 21—An example with $N_{FFD}/N_0 = 11/9$ and N_0 large

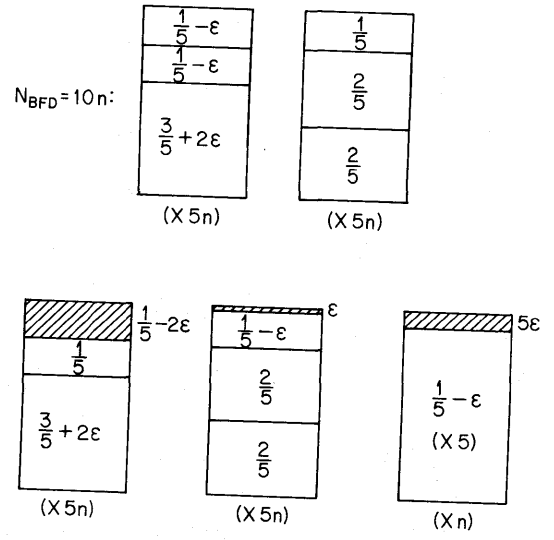

Figure 22—An example with $N_{FFD}/N_{BFD} = 11/10$ and N_0 large

Theorem.[15]

For any $\epsilon>0$, if N_0 is sufficiently large then

$$N_{FFD}/N_0 < 5/4+\epsilon. \tag{12}$$

The example in Figure 21 shows that

$$N_{FFD}/N_0 \geq 11/9 \tag{13}$$

is possible for N_0 arbitrarily large. It is conjectured that the 5/4 in (12) can be replaced by 11/9; this has been established[15] for some restricted classes of α_i.

The preceding remarks also apply, *mutatis mutandis*, to the ratio N_{BFD}/N_0. This is implied by the following somewhat surprising result.

Theorem.[15]

If $\max \alpha_i/\omega^* \geq 1/5$ then $N_{FFD}=N_{BFD}$.

The quantity 1/5 above cannot be replaced by any smaller number as the example in Figure 22 shows.

This example raises the whole question concerning the extent by which the numbers N_{FF}, N_{BF}, N_{FFD} and N_{BFD} may differ among themselves (assuming that N_0 is large). The example in Figure 22 shows that $N_{FFD}/N_{BFD} \geq 11/10$ is possible for arbitrarily large N_0. On the other hand, the example of Figure 23 shows that $N_{BFD}/N_{FFD} \geq 13/12$ is possible for arbitrarily large N_0. These two examples represent the worst behavior of N_{FFD}/N_{BFD} and N_{BFD}/N_{FFD} currently known.

Another algorithm which has been proposed[40] proceeds by first selecting from all the α_i a subset which packs B_1 as well as possible, then selecting from the

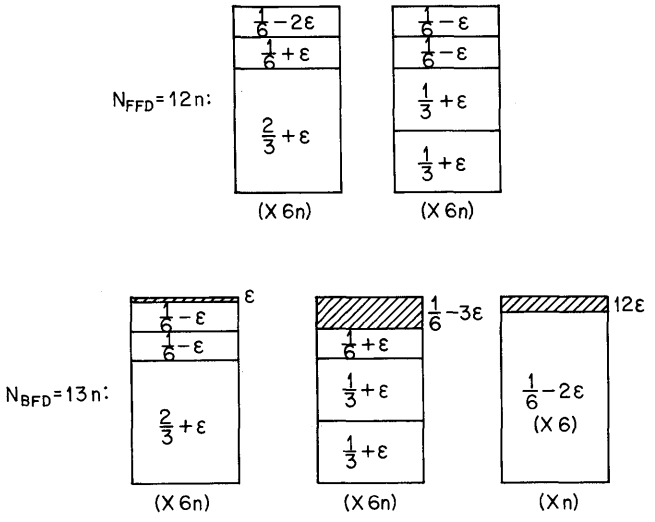

Figure 23—An example with $N_{BFD}/N_{FFD} = 13/12$ and N_0 large

remaining α_i a subset which packs B_2 as well as possible, etc. Although more computation would usually be required for this algorithm than for the first-fit decreasing algorithm it might be hoped that the number $\bar{N}$ of boxes required is reasonably close to N_0. This does *not* have to be the case, however, since examples exist for any $\epsilon > 0$ for which N_0 is arbitrarily large and

$$\bar{N}/N_0 > \sum_{n=1}^{\infty} 1/(2^n-1) - \epsilon. \qquad (14)$$

The quantity

$$\sum_{n=1}^{\infty} 1/(2^n-1) = 1.606695\ldots$$

in (14) is conjectured[15] to be best possible.

Some of the difficulty in proving many of the preceding results and conjectures seems to stem from the fact that a *decrease* in the values of the α_i may result in an *increase* in the number of boxes required. For example, if the weights (760, 395, 395, 379, 379, 241, 200, 105, 105, 40) are packed into boxes of capacity 1000 using the first-fit decreasing algorithm then we find $N_{FFD} = 3$ which is optimal. However, if all the weights are decreased by one, so that now the weights (759, 394, 394, 378, 378, 240, 199, 104, 104, 39) are packed into boxes of capacity 1000 using the first-fit decreasing algorithm, we have $N_{FFD} = 4$ which is clearly not optimal. In fact, the following example shows that N_{FF} can *increase* when some of the α_i are *deleted*. In Figure 24 the list $L = (7, 9, 7, 1, 6, 2, 4, 3)$ is used with the first-fit algorithm to pack boxes of capacity 13, resulting in $N_{FF}(L) = 3$.

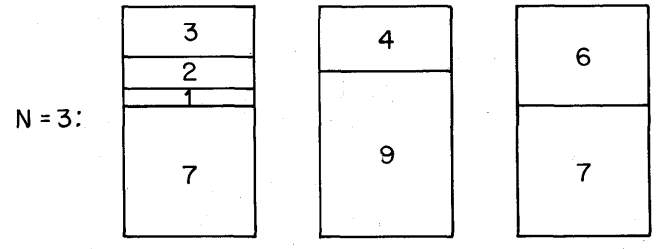

Figure 24—An optimal packing using L

If the number 1 is deleted from L, to form the list $L' = (7, 9, 7, 6, 2, 4, 3)$, then we see in Figure 25 that $N_{FF}(L') = 4$.

DYNAMIC TASK SELECTION

As an alternative to having a fixed list L prior to execution time which determines the order in which

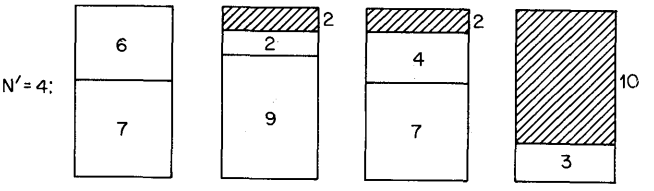

Figure 25—A nonoptimal packing using the deleted list L′

tasks should be attempted, one might employ an algorithm which determines the scheduling of the tasks in a dynamic way, making decisions dependent on the intermediate results of the execution. Unfortunately, no efficient algorithms of this type are known which can prevent the worst possible worst-case behavior.

Perhaps the most natural candidate is the "critical-path" algorithm,[5] which always selects as the next task to be executed, that available task which belongs to the longest† chain of currently unexecuted tasks. This type of analysis forms the basis for many of the project planning techniques which have been developed such as PERT, CPM, etc.[39] Its worst-case behavior can be bad as possible as the example in Figure 26 shows (where $0 < \epsilon < 1$).

If ω_{CP} denotes the finishing time for three processors when the critical path algorithm is used on the example in Figure 26, we have $\omega_{CP} = 2n - 1 - 2\epsilon$. However, the

† Where, as mentioned before, the length of a chain $T_{i1} < \cdots < T_{im}$ is $\sum_k \mu(T_{i_k})$.

optimal solution has $\omega_0 = n$, giving a ratio of

$$\omega_{CP}/\omega_0 = 2 - 1 + 2\epsilon/n \qquad (15)$$

Since ϵ may be chosen arbitrarily close to 0, then ω_{CP}/ω_0 may be arbitrarily close to the previous bound of $2 - 1/n$.

This example also applies to the algorithm which selects as the next task to be executed, that available task for which the sum of the execution times of all its successors is maximal.

It may be true that the critical path algorithm may not have such extreme worst-case behavior when all the $\mu(T_i)$ are nearly equal, although not too much in this direction can be hoped for as the example in Figure 27 shows.

In this example, where n processors are used and all $\mu(T_i) = 1$, $\omega_{CP} = 2n$ is possible depending on how some of the ties are broken. Since $\omega_0 = n+1$ then we obtain a ratio

$$\omega_{CP}/\omega_0 = 2 - 2/(n+1) \qquad (16)$$

which may be the maximum value possible in this case.

CONCLUDING REMARKS

As the reader will have gathered from the preceding discussion, there are certainly more questions than answers available at this point in time. We take this opportunity to comment on several of these questions, indicating what seem to the author to be fruitful directions for further research.

1. What efficient algorithms exist for preventing worst-case behavior in the general multiprocessor problem from approaching the $2 - 1/n$ bound? It certainly seems that just as in the case when there are no precedence constraints between tasks, it should be possible to show in some quantitative sense, that if one is willing to use more complex algorithms, one can be guaranteed of getting closer to the optimum.

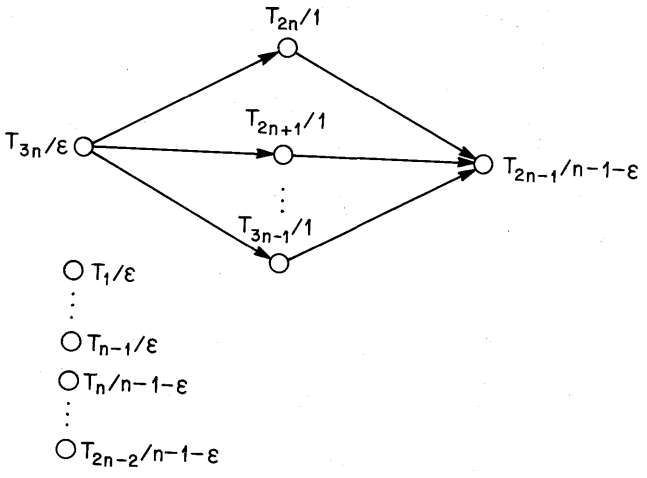

Figure 26—Example which causes worst possible critical path algorithm behavior

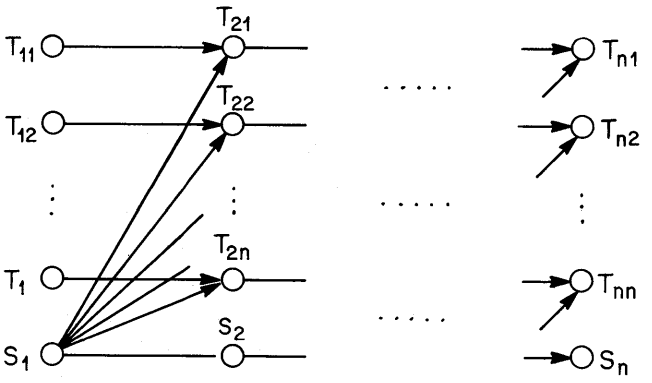

Figure 27—Example which can cause bad critical path algorithm behavior

2. There seems to be little possibility that an efficient† algorithm exists for the determination of optimal schedules for the general multiprocessor problem. Recent work of S. Cook[6] and R. M. Karp (personal communication) helps to clarify some of these issues. They show that a large class of combinatorial problems (one of which is a special case of this problem) are equivalent in this respect, i.e., either they all have efficient algorithms or none do. However, up to now everyone has been singularly unsuccessful in proving the nonexistence of such algorithms. The time seems ripe to remedy this unsatisfactory situation.

3. In the other direction, it seems likely that efficient algorithms should exist for other special cases. For example, good candidates would appear to be the cases $n = 3$, $\mu(T) = 1$ for all T and $n = 2$, $\mu(T) = 1$ or 2 for all T.

4. In reference to the dual (cutting stock) problem of an earlier section, a number of interesting open questions remain, in addition to those already mentioned. For example, one could allow boxes of different capacities and study the behavior of $N_{FF}(L)/N_{FF}(L')$ for a fixed set of weights, as a function of the lists L and L', the ordering of the boxes, the distribution of the capacities and weights, etc.[9] It would also be of interest to examine two-dimensional analogues of these problems in view of the applicability of the results (e.g., see References 16 and 17).

5. Much of the motivation for studying worst-case behavior is derived from the possible insight the results

† In the sense of Edmonds.[8]

may provide for the typical or expected behavior of the system. Very little has been rigorously established in this direction so far although some empirical results are available. For example, in simulation studies involving fairly large task sets, from two to nine processors, and unit task execution times, Manacher[36] reports that in roughly four-fifths of his runs, optimal lists fail to remain optimal when the task execution times are (randomly) slightly perturbed. In other studies, Krone[34] has investigated the typical behavior of several algorithms applied to tasks with no precedence constraints. In particular, he compared the finishing times ω^* and ω' obtained by using the "decreasing" list L^* and by using stabilized pairwise interchanges, respectively. He found that usually $\omega' < \omega^*$ when execution times had a large variance (in spite of the fact that their worst-case behavior is reversed). On the other hand, when the execution times were more nearly equal, ω^* was very good and, in fact, frequently optimal.

In view of the remarkable progress which has occurred in this and other branches of computer science during the past decade, there is little doubt in my mind that the answers to these and many other related questions will be uncovered in the not-too-distant future.

REFERENCES

1 R BELLMAN
Mathematical aspects of scheduling theory
SIAM Jour of App Math 4 1956 pp 168-205

2 W CLARK
The Gantt chart
3rd ed Pitman and Sons London 1952

3 E G COFFMAN JR R L GRAHAM
Optimal scheduling for two-processor systems
To appear Acta Informatica 2 1972

4 E G COFFMAN JR P J DENNING
Operating systems theory
To appear Prentice-Hall 1972

5 R W CONWAY W L MAXWELL L W MILLER
Theory of scheduling
Addison-Wesley Reading 1967

6 S COOK
The complexity of theorem proving
Proc 3rd Annual ACM Symp on Theory of Computing 1971 pp 151-158

7 W L EASTMAN S EVEN I M ISAACS
Bounds for the optimal scheduling of n jobs on m processors
Manag Sci 11 No 2 1964 pp 268-279

8 J EDMONDS
Paths, trees and flowers
Can Jour of Math 17 1965 pp 449-467

9 S EILON N CHRISTOFIDES
The loading problem
Manag Sci 17 No 5 1971 pp 259-268

10 L R FORD JR D R FULKERSON
Flows in networks
Princeton Univ Press Princeton 1962

11 M FUJII T KASAMI K NINOMIYA
Optimal sequencing of two equivalent processors
SIAM Jour of App Math 17 No 3 1969 pp 784-789

12 ———
Erratum
SIAM Jour of App Math 20 No 1 1971 p 141

13 D R FULKERSON
Scheduling in project networks
Proc IBM Scientific Computing Symp on Combinatorial Problems IBM Corp New York 1966 pp 73-92

14 M R GAREY R L GRAHAM
Asymptotic performance bounds on the splitting algorithm for binary testing
To appear in Acta Informatica

15 M R GAREY R L GRAHAM J D ULLMAN
An analysis of some cutting stock algorithms
To appear

16 P C GILMORE R E GOMORY
A linear programming approach to the cutting stock problem
Oper Res 9 1961 pp 849-859

17 ———
A linear programming approach to the cutting stock problem II
Oper Res 11 1963 pp 863-888

18 R L GRAHAM
Unpublished

19 ———
Bounds for certain multiprocessing anomalies
Bell Sys Tech Jour 45 No 9 1966 pp 1563-1581

20 ———
Bounds on multiprocessing timing anomalies
SIAM Jour of App Math 17 No 2 1969 pp 416-429

21 ———
On sorting by comparisons
Computers in Number Theory Ed by A O L Atkin and B J Birch Acad Press New York 1971 pp 263-269

22 A L GUTJAHR G L NEMHAUSER
An algorithm for the line balancing problem
Manag Sci 11 No 2 1964 pp 308-315

23 F HARARY
Graph theory
Addison-Wesley Reading 1969

24 M HELD R M KARP
A dynamic programming approach to sequencing problems
SIAM Jour of App Math 10 No 2 1962

25 M HELD R M KARP R SHARESKIAN
Assembly-line balancing dynamic programming with precedence constraints
Oper Res 11 1963 pp 442-459

26 J HOPCROFT R TARJAN
Planarity testing in V log V steps
Information Processing 1971 Proc of IFIP Congress 71 1971

27 J HOPCROFT R M KARP
An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs
IEEE Conf Record of the Twelfth Annual Symp on Switching and Automata 1971

28 T C HU
Parallel sequencing and assembly line problems
Oper Res 9 No 6 1961 pp 841-848

29 F K HWANG S LIN
Optimal merging of 2 elements with n elements
Acta Informatica 1 1971 pp 145-158

30 S M JOHNSON
Optimal two- and three-stage production schedules with setup times included
Nav Res Log Quart 1 No 1 1954. Also Industrial Scheduling ed by F F Muth and G L Thompson Prentice-Hall Englewood Cliffs NJ 1963

31 J L KELLEY
General topology
Van Nostrand Princeton 1955

32 V KLEE G J MINTY
How good is the simplex algorithm?
Inequalities III ed by O Shisha Acad Press New York 1972 pp 159-175

33 D E KNUTH
The art of computer programming
Vol 3 Addison-Wesley 1972

34 M J KRONE
Heuristic programming applied to scheduling problems
PhD dissertation EE Dept Princeton Univ 1970

35 E L LAWLER
On scheduling problems with deferral costs
Manag Sci 11 No 2 1964 pp 280-288

36 G K MANACHER
Production and stabilization of real-time task schedules
JACM 14 No 3 1967 pp 439-465

37 Ju v MATIJASEVIC
Enumerable sets are diophantine
(In Russian) Doklady 191 1970 pp 279-282

38 R McNAUGHTON
Scheduling with deadlines and loss functions
Manag Sci 6 No 1 1959 pp 1-12

39 J J MODER C R PHILLIPS
Project management with CPM and PERT Reinhold New York 1964

40 R C PRIM
Personal communication

41 E M REINGOLD
Establishing lower bounds on algorithms: a survey
AFIPS Conf Proc 40 1972

42 P RICHARDS
Parallel programming
Report TD-B60-37 Technical Operations Inc 1960

43 J ROBINSON
Diophantine decision problems
Studies in Number Theory ed by W J LeVeque MAA Studies in Math Vol 6 1969

44 J M ROBSON
An estimate of the store size necessary for dynamic storage allocation
JACM 18 No 3 1971 pp 416-423

45 J G ROOT
Scheduling with deadlines and loss functions on k parallel machines
Manag Sci 11 No 3 1965 pp 460-475

46 M H ROTHKOPF
Scheduling independent tasks on parallel processors
Manag Sci 12 No 5 1966 pp 437-447

47 J D ULLMAN
The performance of a memory allocation algorithm
Tech Report No 100 EE Dept Princeton Univ 1971