A New Algorithm for Preemptive Scheduling of Trees

TEOFILO F. GONZALEZ AND DONALD B. JOHNSON

The Pennsylvania State University, University Park, Pennsylvania

ABSTRACT. An algorithm which schedules forests of n tasks on m identical processors in $O(n \log m)$ time, off-line, is given. The schedules are optimal with respect to finish time and contain at most $n - 2$ preemptions, a bound which is realized for all n. Also given is a simpler algorithm which runs in $O(nm)$ time on the same problem and can be adapted to give optimal finish time schedules on-line for independent tasks with release times.

KEY WORDS AND PHRASES: preemptive schedules, minimum finish time, trees, forests, identical processors, uniform processors, efficient algorithms, optimal schedules

CR CATEGORIES. 4.32, 5.25, 5.39

1. *Introduction*

If interruptions are allowed in executing tasks on a set of processors, it is often possible to finish a given set of tasks more quickly than if every task is processed to completion once begun. Such interruptions are called *preemptions*. We consider the general problem of minimizing the finish time for task systems with a treelike precedence structure, attempting to minimize preemptions in the worst case but otherwise ignoring their cost. We deal with the problem when the parameters of all tasks are known in advance and also with an on-line problem with independent tasks. Applications are evident, particularly in computer and communications systems.

This problem was first treated by Muntz and Coffman [13]. Other references relevant to our work are [5, 7, 10–12, 14]. In addition, [2, 3] are of interest as basic references in scheduling theory. The version of this problem in which all tasks are restricted to have unit execution time was originally solved by Hu [8] and has been discussed recently by Davida and Linton [4]. Hu's algorithm schedules trees from leaves to root and therefore bears some resemblance to the more general algorithm of Muntz and Coffman. However, the Muntz and Coffman algorithm does not follow directly from this algorithm. The algorithm of Davida and Linton schedules from root to leaves and therefore bears some resemblance to our algorithms. These authors, however, did not extend their results to treat adequately the problem we solve. As is the case for Hu's rule, application of their scheduling rule to problems with other than unit-time tasks yields suboptimal schedules. Consequently, they propose that a problem with integer execution times be reduced to one with unit execution times by decomposing all tasks into chains of unit-time tasks. Obviously this reduction yields unit-time problems which can be of size exponential in the size of the original input. Thus scheduling can take exponential time and, it can be shown, some schedules will have an exponential number of preemptions which cannot easily be eliminated.

This research was supported in part by the National Science Foundation under Grant MCS 77-21092.

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

1980 ACM 0004-5411/80/0400-0287 $00.75

Our best algorithm schedules forests of n tasks on m identical processors in $O(n \log m)$ time, never producing more than $n - 2$ preemptions. It appears, then, that the interesting comparison is with the Muntz–Coffman algorithm, which runs in $O(n^2)$ time giving schedules with $O(nm)$ preemptions. We make such a comparison in some detail below.

A scheduling problem is specified by a *task system* **P** and an integer $m > 0$ giving the number of identical processors on which **P** is to be serviced. The task system $\mathbf{P} = (\mathcal{T}, \prec, \tau)$, where $\mathcal{T}$ is an indexed set of n tasks, $\prec$ is a partial order on $\mathcal{T}$, and each task T_i in $\mathcal{T}$ has associated with it an *execution time* $\tau(T_i) > 0$ which specifies the total amount of service T_i requires. A schedule for **P** on m processors must provide for each task T_i receiving an amount $\tau(T_i)$ of service from one or more of the m processors in a way that respects $\prec$ and which at any point in time assigns no more than one processor to any task and no more than one task to any processor. We present a precise definition of the term "schedule" later. For the moment it will suffice to say that a schedule will be presented as a set of lists, one for each processor, which gives an assignment of intervals of time to tasks. These lists will be in ascending order on the times and, taken together, must respect the above requirements.

In this paper we will optimize schedules with respect to two criteria, schedule length (the time required to complete all processing under the schedule constructed) and number of preemptions (measured by the number of elements in the schedule lists themselves). As stated above, the partial order $\prec$ is restricted to be a *rooted forest*. We will assume in the discussion of our algorithm that the forest is *initially rooted*, that is, for different tasks T_i, T_j, T_k in $\mathcal{T}$, if $T_i \prec T_k$ and $T_j \prec T_k$, then either $T_i \prec T_j$ or $T_j \prec T_i$. A task T_i is *initial* if for no T_j, $T_j \prec T_i$. A task T_i is *final* if for no T_j, $T_i \prec T_j$. A scheduling problem with exactly one initial task is called a *job*. Thus a nonempty scheduling problem **P** may be partitioned into $r \geq 1$ nonempty jobs which are the trees of the forest defined by **P**. If, for $i = 1, \ldots, r$, $\mathbf{J}_i = (\mathcal{T}_i, \prec_i, \tau_i)$ is the ith job in **P**, then $\mathbf{P} = (\bigcup_{i=1}^{r} \mathcal{T}_i, \bigcup_{i=1}^{r} \prec_i, \bigcup_{i=1}^{r} \tau_i)$, which with a slight corruption of notation we denote as $\mathbf{P} = \bigcup_{i=1}^{r} \mathbf{J}_i$. It is assumed that $\prec$ is presented as a graph without transitive edges so that the space necessary to store $\prec$ is $O(n)$. Choosing $\prec$ to be initially rooted, therefore, is little more than a definitional convenience when schedules are to be constructed off-line. (An off-line algorithm is allowed to receive all of the input for a problem before producing any output.) If $\prec$ were presented as a terminally rooted forest, it could be converted into the form we require in $O(n)$ time. In fact, it is easily seen that a schedule for a terminally rooted problem can be constructed from a schedule for the corresponding initially rooted problem in time proportional to the number of entries in the schedule. We discuss the question of on-line computation later in the paper.

We now proceed to an informal description of the Muntz and Coffman algorithm [13] and of our algorithm so that the basic ideas of their operation may be understood.

Both algorithms rest on an application of the principle of optimality: the given problem **P** is decomposed into an initial problem and a remainder problem, and a rule is given for scheduling the initial problem. The decomposition and the rule have the property that an optimal schedule for **P** may be obtained by following the schedule for the initial problem by any optimal schedule for the remainder problem, provided of course that the first time mentioned in the schedule for the remainder problem is equal to the last time mentioned in the schedule for the initial problem. It may happen that some tasks are split between the initial and remainder problems. The complete algorithm in each case repeatedly applies such a decomposition to the remainder problem and concatenates the schedules produced for each initial problem.

In the Muntz–Coffman algorithm the initial problem is defined by certain leaves of the given forest, the schedule adopted for these leaves, and an event in this schedule. If there are no more than m leaves, the rule is to assign each leaf to a processor. The initial problem is defined by these leaves and the first time at which a leaf will terminate under the rule which schedules each leaf on a processor. Thus the initial problem is comprised of the

leaves, each given an execution time equal to the minimum among them of their original execution times, and an empty precedence constraint. The remainder problem is the given problem with the execution times of the leaves reduced by the amount given to the initial problem.

The Muntz–Coffman algorithm decides the more interesting case, where there are more than m leaves and consequently more tasks are available than there are processors, by scheduling leaf tasks by their current level. The *level* of a leaf task in a terminally rooted forest is the sum of the execution times over all tasks on the path from the task in question to a root. A task's own execution time is included in the sum. The rule is to schedule the m leaf tasks of greatest level, each on one of the m processors. If, however, there are more than m leaf tasks with level greater than or equal to the level of the mth leaf task in some total ordering of leaves by level, then all of these tasks are scheduled as follows: Each leaf with a level greater than the level of the mth leaf has assigned to it a processor. Assume at this point that there are l remaining processors. The remaining k leaves to be scheduled are each assigned to k imaginary processors of reduced speed equal to that of l/k of a true processor. This schedule on a combination of true and imaginary processors is later converted into a schedule on the m true processors.

When there are more than m leaves, the initial problem is defined by the schedule on the true and imaginary processors defined above and the first of two possible events were this schedule to be run on the whole problem: some leaf completes, or the level of some leaf becomes equal to the level of the mth leaf. It is relatively easy to compute the first such event. Partition leaf tasks into three classes: (1) unscheduled, (2) scheduled on an imaginary "slow" processor, and (3) scheduled on a true processor. Keep classes (1) and (3) each in a priority queue, ordered on level. An event is generated by task completion, by the level of the tasks in the class (2) (which are all of equal level) declining to the level of the maximum level task in class (1), or by the level of the lowest level task in class (3) falling to the level of tasks in class (2) (since tasks in class (3) execute faster than those in class (2)). It should be observed for later use that any one original task can generate at most two such scheduling events over the entire execution of the algorithm.

The schedule for an initial problem using imaginary processors can be converted to a schedule on m true processors by taking the k tasks on the k imaginary processors and assigning them preemptively to the $l < k$ unused true processors using the algorithm of McNaughton [11]. This conversion introduces exactly $l - 1$ preemptions for each initial problem when $l > 0$. Since each of the k shorter tasks in an initial problem has the same time requirement on a true processor, it is possible to avoid having any task presented within a schedule for an initial problem also preempted at the point where schedules for initial problems are concatenated.

The execution time of the Muntz–Coffman algorithm is $\Omega(n^2)$, as may be seen from the example in Figure 1. Task times are written in the nodes representing tasks. In this example an initial problem is first defined with $(n + 1)/2$ tasks each of unit execution time. Next, an initial problem with $(n - 1)/2$ tasks is defined, and so forth. Constructing the schedule for each initial problem costs time proportional to the number of tasks in it. The execution time, then, is at least proportional to $m + \sum_{i=m}^{(n+1)/2} i$, which realizes $\Omega(n^2)$ when n is sufficiently larger than m. Since no task can generate more than two scheduling events, it follows that the number of preemptions is $O(nm)$. This bound is realized for the example in Figure 1. Horvath et al. [7] deal with extending the Muntz–Coffman algorithm to arbitrary directed acyclic graphs and to systems with processors of uniformly different speeds. With the exception of problems on two processors or with independent tasks, their algorithms produce suboptimal schedules. We do not deal with such extensions in this paper.

The Muntz–Coffman algorithm schedules by identifying paths of greatest or "critical" length in the remainder problem. Each path from node to root in the given problem becomes critical in this sense at some point. As a comparison with our algorithm will show,

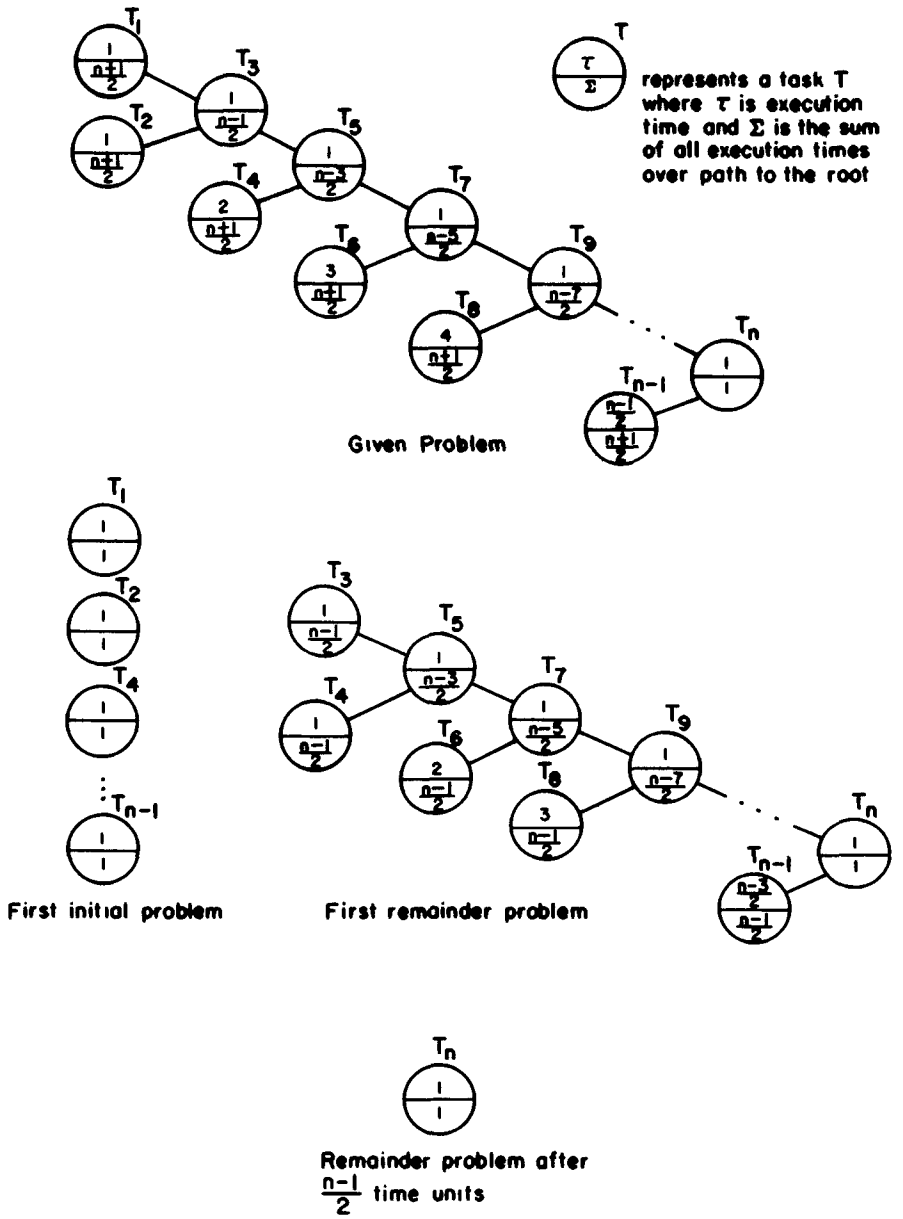

FIG 1. The Muntz–Coffman algorithm is $\Omega(n^2)$

this strategy leads to overspecification of the times at which some of the tasks must be run. Our algorithm succeeds in segregating the tasks into two classes. In one class there is what can be termed the "backbone" of the problem, a superset of those tasks whose start and finish times are fixed in any schedule in which schedule length is minimized. The other tasks can in general be scheduled with some freedom. Our algorithm exploits this freedom to reduce the running time to $O(n \log m)$.

In contrast to the algorithm just described, our algorithm takes the given forest to be initially rooted. Under this assumption we will make the notions of initial and remainder problems more precise. We say that a pair of scheduling problems $(\mathbf{P}', \mathbf{P}'')$ is a *consistent*

decomposition of **P** if

(i) $\mathcal{T} = \mathcal{T}' \cup \mathcal{T}''$;
(ii) $\prec' = \prec$ restricted to $\mathcal{T}'$;
(iii) $\prec'' = \prec$ restricted to $\mathcal{T}''$;
(iv) if $T_i \in \mathcal{T}'$ and $T_j \in \mathcal{T}''$, then (T_j, T_i) cannot be in $\prec$; and
(v) $\tau(T_i) = \tau'(T_i) + \tau''(T_i)$ for all i.

We call **P'** the *initial problem* of the decomposition and **P''** the *remainder problem.* As indicated, our algorithm will repeatedly decompose remainder problems to find a schedule. Proofs of correctness and optimality will rest in part on showing that the algorithm constructs consistent decompositions; hence the above definition. We note for later use that consistent decomposition is an associative operation.

Let the *weight* S_i of a job $\mathbf{J}_i$ be

$$S_i = \sum_{T_j \in \mathcal{T}_i} \tau(T_j) \qquad \text{for} \quad i = 1, \ldots, r.$$

Without loss of generality let the jobs be indexed so that $S_i \geq S_{i+1}$ for $i = 1, \ldots, r-1$. With this ordering, the *critical index* j^* with respect to m identical processors is the greatest j which satisfies $(m-j)S_j > \sum_{i=j+1}^{r} S_i$, or 0 if there is no such j. Jobs $\mathbf{J}_1, \ldots, \mathbf{J}_{j^*}$ are *critical.* The remaining jobs are *noncritical.* In general, there is some freedom allowable in scheduling the tasks in the noncritical jobs, and perhaps no freedom in scheduling the first task in some critical job.

Our "critical weight" algorithm may be stated in simple form as follows.

1 Schedule the initial task of each critical job $\mathbf{J}_i$ on processor i
2 Schedule (preemptively and optimally) the set of noncritical jobs on processors $j^* + 1$ through m
3 Truncate this partial schedule on processors 1 through m at the time of the first of two events: termination of an initial task of a critical job or termination of the schedule constructed on processors $j^* + 1$ through m in step 2, provided this schedule is nonempty. Let the portion of the given problem scheduled up to the point of truncation define an initial problem and the truncated partial schedule be its schedule.
4 Take the schedule for the initial problem and follow it by the schedule obtained by recursive application of the procedure to the remainder problem

A proof of optimality rests on the fact that no task, scheduled when the job of which it is the root is critical, can be scheduled earlier without violating the given precedence constraints. We prove this fact later when the algorithm has been stated precisely enough to do so.

Given this fact, optimality is easy to prove. If the first initial problem generated by the algorithm has fewer than m jobs, then the schedule for the initial problem is optimal. Since no task from the remainder problem can be scheduled before the schedule for the initial problem terminates, the conditions for application of the principle of optimality are satisfied. On the other hand, if the first initial problem has at least m jobs, then we take as the initial problem (for purposes of our proof) the union of all initial problems generated by the algorithm up to but not including the first initial problem with fewer than m jobs. In this case the schedule for the large initial problem so defined is optimal because it keeps all processors busy. Also, any optimal schedule for the remainder problem will begin with tasks all of which belong to critical jobs. By the result mentioned above, no such task can be scheduled earlier. Thus the conditions for application of the principle of optimality are satisfied in all cases and we may conclude that the algorithm produces optimal schedules.

It is easy to implement the algorithm to run in $O(n^2)$ time in the worst case. In the remainder of the paper we show how to achieve the $O(nm)$ and $O(n \log m)$ running times we claim and also discuss the minimization of preemptions. To facilitate the exposition, some of the straightforward but extensive programming details necessary for achieving the running times claimed are presented in appendixes, as is the detailed correctness proof of the basic algorithm. These details are straightforward and could in principle be left to the reader.

2. *Preliminary Results*

Let $\mathbf{M} = \{1, \ldots, m\}$ be a set of $m > 0$ identical processors represented by their indices, and let t_0 and t_f be real numbers satisfying $t_0 < t_f$, the *starting time* and *finishing time*, respectively. For $\mathbf{P}$, a scheduling problem, the set $A = \{([t_1, t_2)_j, T_i)\}$ is an *assignment of* $\mathbf{M}$ *to* $\mathbf{P}$ *in interval* $[t_0, t_f)$ if

(a) for all $([t_1, t_2)_j, T_i) \in A$
  (i) $j \in \mathbf{M}$,
  (ii) $t_0 \le t_1 < t_2 \le t_f$,
  (iii) $T_i \in \mathscr{T}$;
(b) (interval assignments nonoverlapping) for $i, j \in \mathbf{M}$ and $T_k, T_l \in \mathscr{T}$, $([t_1, t_2)_i, T_k)$, $([t_3, t_4)_j, T_l) \in A$ and $(i = j$ or $k = l)$ implies $t_4 \le t_1$ or $t_2 \le t_3$;
(c) (interval assignments conform to τ) for $T_i \in \mathscr{T}$, $\tau(T_i) \ge \sum(t_2 - t_1 \mid ([t_1, t_2)_j, T_i) \in A)$.

A *complete* assignment is one in which equality holds in (c) of the definition for all $T_i \in \mathscr{T}$. An assignment A is *rectangular* if for all $j \in \mathbf{M}$ and for all t, $t_0 \le t \le t_f$, there exists $([t_1, t_2)_j, T_i) \in A$ for which $t_1 \le t \le t_2$. If t_0 and t_f both appear in elements of A, then $t_f - t_0$ is the *length* of A.

An assignment A of $\mathbf{M}$ to $\mathbf{P}$ in interval $[t_0, t_f)$ is *feasible with respect to* $\prec$ if, for all $([t_1, t_2)_i, T_k)$, $([t_3, t_4)_j, T_l) \in A$, $T_k \prec T_l$ implies $t_2 \le t_3$. Our algorithms will generate A in the form of m lists A_i, $i = 1, \ldots, m$, where $([t_1, t_2)_i, T_j) \in A_i$ if and only if $([t_1, t_2)_i, T_j) \in A$. Also, if $([t_1, t_2)_i, T_j)$, $([t_3, t_4)_i, T_k) \in A_i$, then $t_2 \le t_3$ implies $([t_1, t_2)_i, T_j)$ precedes $([t_3, t_4)_i, T_k)$ on list A_i. If A is complete, is feasible with respect to $\prec$, and is represented as described, then A is a *schedule.* In such representations we will omit the interval subscripts, writing element $([t_1, t_2)_i, T_j)$ as $([t_1, t_2), T_j)$ if the element is known to be on list A_i.

We have defined a preemption informally as an interruption in the service of a task. The *number of preemptions* in an assignment A in which n distinct tasks appear is $card(A) - n$, where $card(X)$ is the number of elements in set X.

If $\mathbf{P}$ is a scheduling problem which satisfies $mS_j \le \sum_{i=1}^{r} S_i$ for $j = 1, \ldots, r$, then $\mathbf{P}$ is said to be *noncritical in* $\mathbf{M}$. If $\mathbf{P}$ is noncritical, then for every t_0 there exists a feasible rectangular complete assignment of $\mathbf{M}$ to $\mathbf{P}$ for the interval $[t_0, t_0 + \sum_{i=1}^{r} S_i/m)$ [11]. This result, which we have alluded to in the introduction, employs a simple "bin packing" construction which we now describe informally. Take the jobs in some order and, beginning with processor 1, allocate time on this processor for the first job, beginning at t_0. The tasks of the job are themselves allocated one at a time at the earliest free time in an order consistent with $\prec$. Continue to allocate jobs on the first processor at the earliest free time until a job is encountered for which the remaining time in the interval is insufficient. This job is split between the first and second processors so that tasks of this job (and possibly a part of a task) occupy all of the remaining time on the first processor. The remaining tasks (and possibly a fraction of a task) are allocated time on the second processor, starting at t_0. The tasks given to the second processor are chosen to come first in some order consistent with $\prec$, so that $\prec$ is not violated between processors 1 and 2. The procedure is then continued on processor 2.

This procedure treats each job essentially as if it were a single task, but of course the final schedule must be presented in terms of the tasks of the given problem. These requirements are met by *flattening* jobs, that is, finding a total order for job $\mathbf{J}_i$ consistent with $\prec_i$. In particular, for a scheduling problem $\mathbf{P}$ the list $N = \bigcup_{i=1}^{r}\{(S_i, ((\tau(\mathrm{T}_1), \mathrm{T}_1), \ldots, (\tau(\mathrm{T}_{n_i}), \mathrm{T}_{n_i})))\mid$ without loss of generality $\mathscr{T}_i = \{T_1, \ldots, T_{n_i}\}$ and $T_j \prec_i T_k$ implies $j < k$ for $T_j, T_k \in \mathscr{T}_i\}$ is a *flattened list for* $\mathbf{P}$. As the following lemma states, the ideas of [11] can be embodied in a slightly more general procedure which accepts a scheduling problem in the form of a flattened list, truncates this schedule at a cutoff time t_c to define an initial problem, and returns the truncated schedule in an assignment A and the remainder problem in a list.

LEMMA 2.1. *Let* $\hat{\mathbf{M}} \subseteq \mathbf{M} = \{1, \ldots, m\}$, $\hat{m} = card(\hat{\mathbf{M}})$, $t_c > t_0$, *and A be any assignment of* **M**. *If* **P** *is a scheduling problem satisfying* $\hat{m}S_i \leq \sum_{j=1}^{r} S_j$, $i = 1, \ldots, r$, *and N is a flattened list for* **P**, *then there exists a procedure RECTANGLE for which*

(*i*) *RECTANGLE*$(A, N, \hat{\mathbf{M}}, t_0, t_c) = (A', N'')$ *is well defined*;

(*ii*) *there exists a consistent decomposition* $(\mathbf{P}', \mathbf{P}'')$ *of* **P** *for which* $A' - A$ *is a complete feasible rectangular assignment of* $\hat{\mathbf{M}}$ *to* $\mathbf{P}'$ *in* $[t_0, min\{t_c, t_0 + \sum_{i=1}^{r} S_i/\hat{m}\})$ *and* N'' *is a flattened list for* $\mathbf{P}''$;

(*iii*) $(s, Q) \in N''$ *implies* $s \leq t_0 + \sum_{i=1}^{r} S_i/\hat{m} - t_c$.

PROOF. Parts (i) and (ii) follow from [11] and an examination of the procedure RECTANGLE shown in Appendix A. The ideas have been outlined in the informal discussion above. We omit further details. Part (iii) follows from the fact that if N'' is nonempty, it contains exactly the jobs and parts thereof which would be scheduled in the interval $[t_c, t_0 + \sum_{i=1}^{r} S_i/\hat{m})$ by the procedure of [11]. □

It should be noticed that when t_c is sufficiently large, RECTANGLE reduces to the procedure of [11] as applied to noncritical scheduling problems.

Certain sets employed in RECTANGLE and the other algorithms of this paper are lists on which certain primitive operations are defined. The operation *pop* deletes and returns the last element inserted in the list by the operation *push*. The element itself is the *head* of the list. In other words, for a list X and an element x, $pop(push(x, X)) = x$ and $head(push(x, X)) = x$. We will also use pointers to list elements and employ a pointer to the head of a list as a pointer to the list itself. Thus if p is a pointer, then $elem(p)$ is the list element pointed to by p. For lists X and Y and pointers p and q, where $elem(p) = head(X)$ and $elem(q) = head(Y)$, the result of the assignment $Y \leftarrow X$ will be $elem(p) = head(Y)$.

3. *The "Critical Weight" Algorithm*

As observed above, any noncritical scheduling can be scheduled by the procedure RECTANGLE. However, when the noncriticality condition does not hold, this procedure will fail in general to produce a schedule of minimum length. Noncriticality holds in no case when the number of jobs r satisfies $r < m$, and noncriticality may not hold when $r \geq m$. In the introduction the critical index for a scheduling problem **P** was defined as $j^* = \max\{0, j |$ for all $i = 1, \ldots, j$, $(m - i)S_i > \sum_{k=i+1}^{r} S_k\}$ where, without loss of generality, $S_i \geq S_{i+1}$ for $i = 1, \ldots, r - 1$. A job $\mathbf{J}_i$ is critical in **P** (for given m) if $i \leq j^*$. Otherwise the job is noncritical in **P**. Our strategy is to schedule, one to a processor, part of the initial task of each critical job up to a cutoff time t_c. Such an assignment is always possible since, as may be seen, $j^* < m$. The remaining jobs (if any) are scheduled according to the procedure RECTANGLE on the unused processors, of which there is at least one when $r \geq m$. The cutoff time t_c is used as a parameter to RECTANGLE. Consequently, a schedule is completed for an interval ending at t_c, and the remainder problem is scheduled by a reapplication of the rule at time t_c.

We now give in detail the algorithm which embodies this strategy, establish its correctness, and then (in Section 4) show how it can be implemented to run in $O(n \log m)$ time. A simpler variant, which runs in $O(nm)$ time, is given first. This variant is of use for expository purposes and also because it leads to an $O(nm)$ on-line algorithm for problems with independent tasks and release times, a problem which Horn solved [6] with an algorithm which runs in $O(n^2)$ time.

At the start of each iteration of the algorithm, there will exist a consistent decomposition $(\mathbf{P}', \mathbf{P}'')$ of **P** for which $\mathbf{P}'$ will be scheduled in A and $\mathbf{P}''$ will not. For reasons of efficiency it is important to store each job $\mathbf{J}_i''$ of $\mathbf{P}''$ in one of two forms, the flattened list elements of the form (s_i, Q_i), already defined, and elements of the form (s_i, u_i, T_i). In the former case $s_i = S_i''$, but in the latter case $s_i = S_i'' + t$ for the "current" time t. To be precise, for $t \geq 0$ and a scheduling problem **P**, a list I is a *list for* **P** *at* t if $I = \bigcup_{i=1}^{r}\{(s_i, Q_i)$

or $(s_i, u_i, T_i) \mid (s_i, Q_i)$ is a flattened list for $\mathbf{J}_i$ and (s_i, u_i, T_i) satisfies $s_i = S_i + t$, $u_i = \tau(T_i) + t$, and T_i is initial in $\mathbf{J}_i\}$. We assume there is available from a traversal of $\mathbf{P}$ a function $\sigma: \mathscr{T} \to \mathscr{R}$, where $\sigma(T_i) = \tau(T_i) + \sum(\tau(T_j) \mid T_i \prec T_j)$ for $T_i \in \mathscr{T}$.

Algorithm CRITICAL_WT(**P**, *m*)

1. $C \leftarrow \varnothing$;
2. $N \leftarrow \varnothing$;
3. $t \leftarrow 0$;
4. $S \leftarrow 0$;
5. $L \leftarrow \bigcup_{i=1}^{r} \{(\sigma(T_i), \tau(T_i), T_i) \mid T_i$ *is initial in* $\mathbf{J}_i\}$,
6. **for** $i \leftarrow 1$ **until** m **do** $A_i \leftarrow \varnothing$ **endfor**
7. **while** *card*$(C \cup L \cup N) > 0$ **do**

H1 =
- (a) There exists a consistent decomposition $(\mathbf{P}', \mathbf{P}'')$ of $\mathbf{P}$ for which
  - (i) A is a complete feasible assignment of $\mathbf{M}$ to $\mathbf{P}'$ in $[0, t)$,
  - (ii) $C \cup L \cup N$ is a list for $\mathbf{P}''$ at t,
  - (iii) $S = \sum(s_i \mid (s_i, Q_i) \in N)$,
- (b) For $(s_i, u_i, T_i) \in C \cup L$ there exists $\{T_{i_1}, \ldots, T_{i_k} = T_i\}$ where T_{i_1} is initial in $\mathbf{J}_{i_1} \in \mathbf{P}$ and, for $j = 1, \ldots, k-1$, $T_{i_j} \prec T_{i_{j+1}}$ and $\sum_{j=1}^{k} \tau(T_{i_j}) = u_i$.

//Partition jobs with respect to j^* and determine cutoff time $t + \Delta$//

8 $(C, L, N, S, \Delta) \leftarrow SPLIT(C, L, N, S, t)$;

H2 =
- (a) There exists a consistent decomposition $(\mathbf{P}' \cup \mathbf{P}'')$ of $\mathbf{P}$ for which
  - (i) A is a complete assignment of $\mathbf{M}$ to $\mathbf{P}'$ in $[0, t)$,
  - (ii) $C \cup N$ is a list for $\mathbf{P}''$ at t,
  - (iii) $S = \sum(s_i \mid (s_i, Q_i) \in N)$,
  - (iv) $(s_i, u_i, T_i) \in C$ iff $\mathbf{J}_i''$ is critical, where T_i is initial in $\mathbf{J}_i''$;
- (b) $L = \varnothing$;
- (c) $\Delta = \min_{\text{nonzero values}} (\{u_i - t \mid (s_i, u_i, T_i) \in C\} \cup \{S/(m - card(C))\})$.

//Extend schedule of critical tasks to $t + \Delta$//

9 **for** $(s_i, u_i, T_i) \in C$ **do** *push*($([t, t + \Delta), T_i), A_i$) *where, wlog,* $C = \{(s_1, u_1, T_1), \ldots, (s_k, u_k, T_k)\}$ **endfor**

//Extend schedule of noncritical tasks to $t + \Delta$//

10 **if** $S > 0$ **then** $(A, N) \leftarrow RECTANGLE(A, N, \{card(C) + 1, \ldots, m\}, t, t + \Delta)$ **endif**

11 **if** $S > 0$ **then** $S \leftarrow S - \Delta(m - card(C))$ **endif**

//Delete from C any jobs with initial tasks completed at $t + \Delta$ and put the successor jobs in L//

12 **for** $(s_i, u_i, T_i) \in C$ *satisfying* $u_i - (t + \Delta) = 0$ **do**
  $C \leftarrow C - \{(s_i, u_i, T_i)\}$;
  for T_j *satisfying* $T_i \prec T_j$ *and, for no* T_k, $T_i \prec T_k \prec T_j$ **do**
    $L \leftarrow L \cup \{(\sigma(T_j) + t + \Delta, \tau(T_j) + t + \Delta, T_j)\}$
  endfor
endfor

//Update time//

13. $t \leftarrow t + \Delta$

endwhile

14 **return**(A, t)

end *CRITICAL_WEIGHT*

Figure 2 shows an example of a scheduling problem with 24 tasks. Algorithm CRITICAL_WT gives the schedule shown schematically in Figure 3 when $m = 3$. This schedule contains 12 preemptions, 9 of which can be removed easily by further processing to be discussed later.

Two assertions, H1 and H2, are embedded in the algorithm. It is easily verified that a procedure SPLIT exists which returns an output satisfying H2 when supplied an input satisfying H1. We discuss later an implementation which runs in $O(n \log m)$ time over the entire execution of the algorithm. We defer further discussion of SPLIT until that time.

LEMMA 3.1. *Assertion H1 is invariant over every iteration of the loop at step 7 of Algorithm CRITICAL_WT.*

The proof of Lemma 3.1 is given in Appendix B.

LEMMA 3.2. *Algorithm CRITICAL_WT executes at most n iterations of the loop at step 7 on any scheduling problem* **P** *with* $m > 0$.

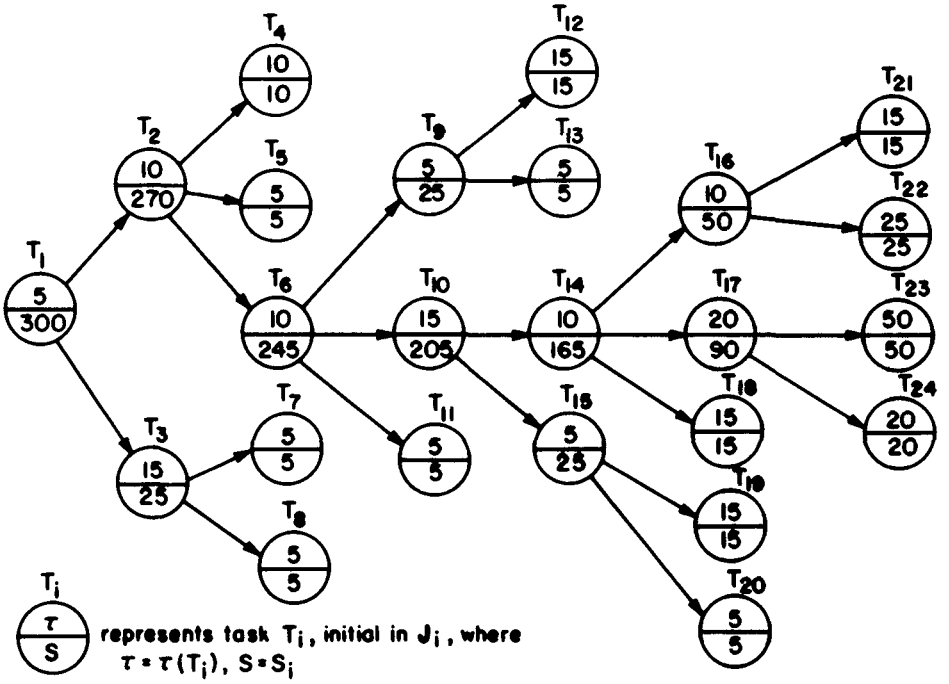

FIG. 2 An example scheduling problem

PROOF. Consider at the start of an iteration of the loop at step 7 the consistent decomposition $(\mathbf{P}', \mathbf{P}'')$ of $\mathbf{P}$. If $\mathbf{P}'' = (\mathscr{T}'', \prec'', \tau'')$, we notice that at least one task is deleted from $\mathscr{T}''$ during the ensuing iteration. Since $\mathscr{T}'' = \emptyset$ implies $C \cup L \cup N = \emptyset$, termination must occur in at most n iterations. □

THEOREM 3.1. *Given a scheduling problem* $\mathbf{P}$ *and* $m > 0$, *CRITICAL_WT*$(\mathbf{P}, m)$ *is well defined. If CRITICAL_WT*$(\mathbf{P}, m) = (A, t_f)$, *then* A *is a schedule for* $\mathbf{P}$ *on* $\mathbf{M}$ *in interval* $[0, t_f)$ *and for no* $t < t_f$ *does there exist any schedule for* $\mathbf{P}$ *on* $\mathbf{M}$ *in* $[0, t)$. *The number of preemptions in* A *is less than or equal to* $2nm - 4n - m + 3$ *for* $m \geq 2$.

PROOF. From Lemma 3.2 we have that $C \cup L \cup N = \emptyset$ after at most n iterations. Since H1 must hold after line 13 is executed for the last time (Lemma 3.1), it follows that A is a complete assignment of $\mathbf{M}$ to $\mathbf{P}$ in $[0, t_f)$. Examination of the algorithm verifies that A is presented in the form of a schedule.

The proof of optimality follows from the discussion given in the introduction and the invariance of part (b) of assertion H1 established in Lemma 3.1.

In any interval $[t, t + \Delta)$ other than $[t, t_f)$, preemptions may be generated in both critical and noncritical jobs. The way the time increment Δ is chosen may cause as many as j^* preemptions on processors $\{1, \ldots, j^*\}$. On processors $\{j^* + 1, \ldots, m\}$ it is possible that there will be $m - j^* - 1$ preemptions internal of the interval $[t, t + \Delta)$ and $m - j^* - 1$ preemptions at the end of the interval if RECTANGLE produces $N \neq \emptyset$ (in which case only $j^* - 1$ preemptions are possible on the first j^* processor). Notice that one preemption at time $t + \Delta$ can be recovered in the next interval if RECTANGLE schedules first the last job it puts into N on the previous iteration and reverses from iteration to iteration the order in which it schedules free processors.

The maximum number of preemptions chargeable to $[t, t + \Delta)$ for $t + \Delta < t_f$ is $\max_{0<j^*<m} \{j^* - 1 + 2(m - j^* - 1)\} = 2m - 4$, where there are $j^* - 1$ preemptions possible on $\{1, \ldots, j^*\}$ and $2(m - j^* - 1)$ possible on $\{j^* + 1, \ldots, m\}$. Notice that $t + \Delta < t_f$ requires that $j^* \geq 1$ and $m \geq 2$. Otherwise only one interval occurs. In the

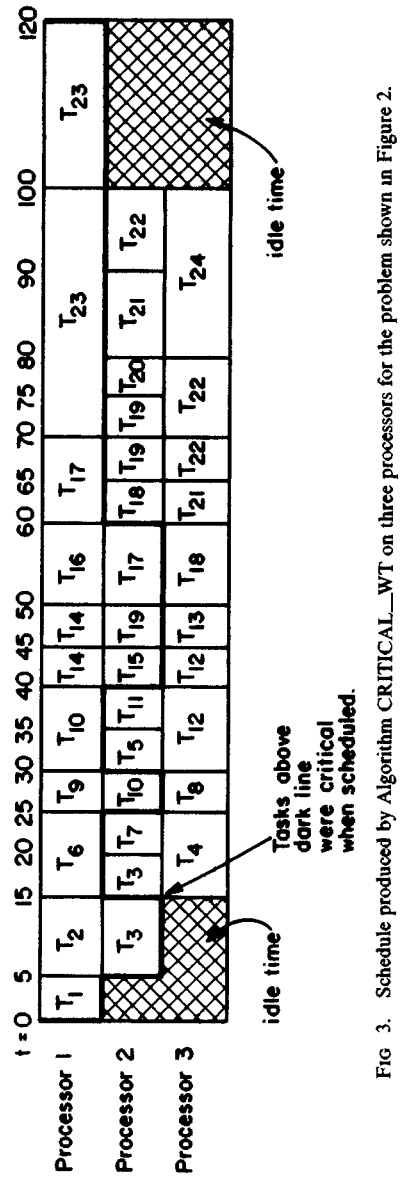

FIG 3. Schedule produced by Algorithm CRITICAL_WT on three processors for the problem shown in Figure 2.

last interval there can be at most $m - j^* - 1$ preemptions for $j^* \geq 0$. Combining we get $(n-1)(2m-4) + m - 1 = 2nm - 4n - m + 3$. For $m = 2$, this bound reduces to 1. □

To obtain the bound of $O(nm)$ on running time, we confine our attention to steps 8, 9, 10, and 12. All other steps can easily be seen to require $O(n)$ time over the entire execution. Let the set $C = \{(s_i, u_i, T_i)\}$ be kept in two binary heaps [1], one ordered on s_i and one on u_i. After execution of step 8, $card(C) < m$. With care, a bound of m can be maintained throughout execution. We notice that each deletion of an element from C in step 12 can be charged to a task. Thus over the entire execution of the algorithm, step 12 will cost $O(n \log m)$. Continuing the analysis, it was established in the proof of Lemma 3.1 that no element is ever moved by SPLIT from N to C. Thus SPLIT can be implemented to first merge L into C, discarding smallest elements from C whenever $card(C) = m$. Then j^* can be found by the further moving of smallest elements from C to N. Each such movement is charged to a unique task. The weight S of N can be computed as the calculation proceeds. The heap ordered on u_i may be used to discover Δ. Altogether, step 8 will run in $O(n \log m)$ time.

The costly steps are 9 and 10. It is easily seen that step 9 will require $O(nm)$ time in the worst case. Each execution of step 10 is $O(card(N))$. Let N_k be the set N before step 10 on the kth iteration, and let N'_k be the set N output in step 10 on the kth iteration. It is clear that $N_k = N'_{k-1} \cup \{\text{elements rejected from } C \text{ in step } 8\}$. Since $card(N') < 2m$ and the number of elements rejected from C cannot exceed n in total, $\sum_{k=1}^{n} N_k < 2mn + n$. Thus step 10 contributes $O(nm)$ time over all iterations. This analysis supports the following result.

THEOREM 3.2. *Algorithm CRITICAL_WT can be implemented to run in $O(nm)$ time.*

As is evident from Figure 3, some preemptions can in general be eliminated from schedules produced by Algorithm CRITICAL_WT. The elements of A can be collected in $O(nm)$ time into a list in which any pair $([t_1, t_2)_i, T_k)$ and $([t_2, t_3)_j, T_k)$ will be adjacent. Segments of the lists A_i and A_j can then be swapped so that all preemptions in which a task T_k is preempted and resumed at the same time t_2 occur on the same processor. List elements can then be coalesced to recover all such preemptions. In fact, this process can be embedded in Algorithm CRITICAL_WT at an increased cost of only a constant factor.

In the next section we focus attention on steps 9 and 10 in order to cut the running time to $O(n \log m)$. In the faster algorithm the easily recoverable preemptions discussed will not be generated in the first place. A salient feature of Algorithm CRITICAL_WT, which we now discuss, is its adaptation to on-line computation. Let there be given a scheduling problem **P** in which the tasks are independent ($\prec = \emptyset$) but each task $T_i \in \mathcal{T}$ has a release time $\rho(T_i)$. An assignment A of **M** to **P** is *feasible with respect to* ρ if $([t_1, t_2), T_i) \in A$ implies $t_1 \geq \rho(T_i)$ for all $T_i \in \mathcal{T}$. As mentioned, an algorithm is known which solves this problem in $O(n^2)$ time [6]. This algorithm is off-line in the sense that the entire problem must be known before the interval between the first two release times can be scheduled.[1] Our algorithm can be adapted to solve this problem on-line in $O(nm)$ time. We assume that the given scheduling problem is presented on-line in order of release time. Algorithm CRITICAL_WT operates as before except that certain operations wait for parts of the schedule to be executed before they are performed. In particular, the algorithm waits to execute step 11 until the on-line time advances to $t + \Delta$, since it is possible that Δ will be redefined on-line by the occurrence of a release. If the algorithm is waiting to execute step 11 and a release occurs, Δ is immediately redefined to let $t + \Delta$ equal the current on-line time, and the results of steps 8–10 are adjusted to agree with this new value. In step 12, tasks are deleted from C as before, but L is constructed from the new tasks released at time $t + \Delta$. Correctness, optimality, and run-time analysis are essentially as before. The number of preemptions does not exceed $2nm - 2n - m + 2$. If our algorithm is used to solve this problem off-line, it may be necessary to charge $O(n \log n)$ time to sort $\mathcal{T}$ by ρ. The

[1] We are also aware of an $O(n \log nm)$ off-line algorithm of Sahni [14]

analogous problem on independent tasks with due times can be solved off-line within a time bound of the same order by transforming the problem into a release time problem and reversing the schedule found.

4. *An O(n log m) Algorithm*

In the analysis of Algorithm CRITICAL_WT, steps 9 and 10 were identified as the only steps requiring more than $O(n \log m)$ time. We now show how to modify the algorithm to bring these steps within the desired bound. What we will do is postpone the action of step 10 and simply accumulate all the "new" members of N each time step 10 would be executed. Associated with the jobs which would be scheduled will be the time t at which they first would have entered N. This time will be called the *release time* for the given job. Then when it first happens that $S = 0$, all jobs saved will be scheduled in the interval where S was greater than zero. It will always be possible to schedule tasks in the free time in which CRITICAL_WT would have scheduled them in step 10.

The excessive time spent in step 9 arises from preempting initial tasks of critical jobs at each time t when, in fact, these tasks may appear continuously over a larger interval in the completed schedule. We show how to keep these tasks "on the same processor" and not generate preemptions in the first place. Doing so involves two innovations. When an initial task T_i from C is scheduled on A_j at time $t = t_1$, the assignment will be incompletely specified. The element placed on A_j will be $([t_1, \infty), T_i)$. When t equals the termination time, either because $\tau(T_i)$ is exhausted or $\mathbf{J}_i$ becomes noncritical, the symbol ∞ is replaced with t. For purposes of efficiency the set C is partitioned into two sets $C = C_a \cup C_d$, where C_a is the set of *active* critical jobs and C_d is the set of *dormant* critical jobs. When an element $([t_1, \infty), T_i)$ is first placed on A_j, $(s_i, u_i, T_i) \in C_a$. The element (s_i, u_i, T_i) is then moved to C_d where it remains until $([t_1, \infty), T_i)$ is terminated. The crucial point is that only the elements of C_a need be considered in step 9.

We now present the modifications in sufficient detail to establish correctness and prove a bound of $O(n \log m)$ on running time. It will no longer be possible always to schedule the initial tasks of the j^* critical jobs on the first j^* processors, so we will keep the indices of the processors available to RECTANGLE on a list Z. (Later RECTANGLE is dispensed with, but the list Z will still be needed.) Initially $Z = \{1, \ldots, m\}$. Processors become unavailable when assigned in step 9. Under the modifications to be shown, a processor will remain assigned until either the task scheduled is found in step 12 to terminate or the job to which it belongs becomes noncritical in step 8. Also introduced is the set Z_{BRK} which contains exactly those processors on which a critical task begins or terminates at time t. The modifications are made in two stages in order to facilitate proof of correctness.

Our first modification will change steps 8–12 to reduce the running time of step 9. The changes yield the following loop at step 7. It is assumed that initialization $Z \leftarrow \{1, \ldots, m\}$ and $Z_{BRK} \leftarrow \varnothing$ is performed.

```
7     while card(C_a ∪ C_d ∪ L ∪ N) > 0 do
          //Partition jobs with respect to j* and determine cutoff time t + Δ//
8(a)      (C_a, C_d, L, N_NEW, S, Δ, Z, Z_BRK) ← SPLIT(C_a, C_d, L, S, t, Z, Z_BRK),
8(b)      N ← N ∪ N_NEW,
          //Extend schedule of critical tasks to t + Δ//
9         for (s, u, T, ·) ∈ C_a do
              i ← pop(Z),
              push(([t, ∞), T), A_i);
              let p satisfy elem(p) = head(A_i),
              C_a ← C_a − {(s, u, T, ·)},
              C_d ← C_d ∪ {(s, u, T, p)},
              Z_BRK ← Z_BRK ∪ {i}
          endfor
          //Extend schedule of noncritical tasks to t + Δ//
10.       if S > 0 then (A, N) ← RECTANGLE(A, N, Z, t, t + Δ) endif
```

11 **if** $S > 0$ **then** $S \leftarrow S - \Delta(m - card(C_a \cup C_d))$ **endif**
//Delete from $C_a \cup C_d$ any jobs with initial tasks completed at $t + \Delta$ and put successor jobs in L//
12. $(L, C_d, Z, Z_{BRK}) \leftarrow CLOSE(C_d, t, \Delta, Z, Z_{BRK})$,
13 $t \leftarrow t + \Delta$
endwhile

The procedure SPLIT operates as before, with the following embellishment: When a noncritical job is removed from C_d to be put into N, it is necessary to complete its entry on the list schedule with the finish time t. The pointer field p in the element $(s, u, T, p) \in C_d$ facilitates this operation. Completing such an entry frees a processor, so this event is recorded in the set Z_{BRK} and the processor is added to the set of free processors Z. Step 9 then proceeds essentially as before to generate new elements for the schedule list from the newly critical jobs, all of which are in C_a. To do this for $(s, u, T, \cdot)$ in C_a, a free processor i is obtained from Z, the element $([t, \infty), T)$ is pushed onto list A_i, and (s, u, T, p) is put in C_d, where p points to the element just pushed onto A_i. The commitment of processor i is recorded by deleting its index from Z and entering the index in Z_{BRK}. Steps 10 and 11 are unchanged.

Step 12 is now implemented with a procedure CLOSE which operates only on C_d since C_a is empty when CLOSE is called. Jobs in C_d with zero execution time remaining for their initial tasks are processed as in step 12 of CRITICAL_WT, but, in addition, schedule entries are completed with time t and the freeing of processors is recorded as in the new procedure SPLIT.

It follows from the above discussion that steps 8(a) and 8(b) can satisfy the input–output requirements defined by H1 and H2 if $C_a \cup C_d$ is taken as C and the additional pointer field in elements of $C \cup L$ is ignored. Reference to the realization of SPLIT shown in Appendix C allows this assertion to be verified in detail. It should be observed that the set N is not needed as an input to SPLIT. The variable S contains sufficient information on the contents of N.

It may be shown by induction that if $(s, u, T, \cdot) \in C_d$ before step 9, then there is at that moment an element $([t_1, \infty), t)$ in A. Thus it is correct for step 9 to reference only elements of C_a. The call in step 12 is not in fact restricted to a proper subset of C because, at this point, $C_d = C_a \cup C_d$. The effect of putting "open-ended" elements of the form $([t, \infty), T)$ into A in step 9 is to permute in each iteration the indices of the processors, so that the several assignments which CRITICAL_WT in general makes to one task in contiguous time intervals but on several processors are coalesced into one element in A on one processor. The processors which are free to RECTANGLE in step 10 are recorded in Z. Freed processors are put into Z in steps 8 and 12 and are removed in step 9, as described above. It can be shown by induction that the modified algorithm does indeed schedule critical tasks in the same time intervals as does CRITICAL_WT, and that Z contains exactly those $m - j^*$ processors which are available to RECTANGLE in step 10. In order to complete a proof of invariance of H1 under these modifications, it is necessary only to substitute the current value of t for each occurrence of ∞ in elements of A. The details are a straightforward parallel of the proof of Theorem 3.1 and will not be discussed further. We notice that $p \neq \Lambda$ in any $(s, ((\Delta, T), \ldots), p) \in N$ is a pointer to an element $([t_1, t_2), T)$ in A, where $t_2 \neq \infty$. These pointers give us the potential to recover preemptions generated when an initial portion of a task is to be scheduled as part of a noncritical job.

As we have just argued, the above modifications preserve optimality of the schedule produced. The complexity arguments already given for steps 8 and 10–12 remain unchanged, as may be verified in detail by reference to Appendix C where realizations of SPLIT and CLOSE are shown. We notice that confining the domain of step 9 to C_a reduces the total time spent in step 9 to $O(n)$ because no task repeats in C_a. The only step which still exceeds the desired bound of $O(n \log m)$ is step 10.

We now replace step 10 with a statement which will save noncritical jobs for scheduling later.

```
    //Save new noncritical jobs//
10  for (s, Q, p) ∈ N_NEW do push((t, s, Q, p), R) endfor
```

To schedule the tasks in the list R at the times when N would normally become empty by the action of RECTANGLE, statement 14 is added.

```
    //Initiate scheduling of accumulated noncritical jobs//
14. if S = 0 then
        (A, R, Z) ← PACK(A, R, Z, t);
        Z_BRK ← ∅
    endif
```

We notice that N becomes vestigial under the modifications. What remains to be shown is that the deferred scheduling of noncritical jobs can be realized to run in $O(n \log m)$ time overall. Of course, correctness is trivial if complexity is not an issue. The procedure PACK could simply mimic the action of RECTANGLE at each release time when jobs were put on the list R. It would suffice to insert at the appropriate places in R the sets Z of available processors. Our plan, however, is to schedule jobs from later to earlier times in a way which respects release times but introduces fewer preemptions. It in fact will not be possible to obtain our time bound if the sets Z are stored in R. Just storing them would cost $O(nm)$. Instead, we store Z_{BRK}. The complete algorithm, FAST_CRITICAL_WT, is as shown below.

```
Algorithm FAST_CRITICAL_WT(P, m)
1(a)  C_a ← ∅;
1(b)  C_d ← ∅;
1(c)  Z ← ∅;
1(d). Z_BRK ← ∅;
1(e)  for i ← m by −1 until 1 do push(i, Z) endfor
2.    R ← ∅,
3.    t ← 0,
4.    S ← 0;
5.    L ← ∪_{i=1}^{r} {(σ(T_i), τ(T_i), T_i, Λ) | T_i is initial in J_i};
6     for i ← 1 until m do A_i ← ∅ endfor
7     while card(C_a ∪ C_d ∪ L) + S > 0 do
          //Partition jobs with respect to J* and determine cutoff time t + Δ//
8(a)      (C_a, C_d, L, N_NEW, S, Δ, Z, Z_BRK) ← SPLIT(C_a, C_d, L, S, t, Z, Z_BRK),
8(b)      if R ≠ ∅ then push(Z_BRK, R) endif
8(c).     Z_BRK ← ∅;
          //Extend schedule of critical tasks to t + Δ//
9.        for (s, u, T, ·) ∈ C_a do
              i ← pop(Z);
              push(([t, ∞), T), A_i),
              let p satisfy elem(p) = head(A_i),
              C_a ← C_a − {(s, u, T, ·)},
              C_d ← C_d ∪ {(s, u, T, p)},
              Z_BRK ← Z_BRK ∪ {i}
          endfor
          //Save new noncritical jobs//
10        for (s, Q, p) ∈ N_NEW do push((t, s, Q, p), R) endfor
11        if S > 0 then S ← S − Δ(m − card(C_a ∪ C_d)) endif
          //Delete from C_d any jobs with initial tasks completed at t + Δ and put the successor jobs in L//
12        (L, C_d, Z, Z_BRK) ← CLOSE(C_d, t, Δ, Z, Z_BRK),
          //Update time//
13        t ← t + Δ,
          //Initiate scheduling of accumulated noncritical jobs//
14        if S = 0 then
              (A, R, Z) ← PACK(A, R, Z, t),
              Z_BRK ← ∅
          endif
      endwhile
15    return (A, t)
end FAST_CRITICAL_WT
```

It is easy to show by induction that the list R is of the following form when PACK is called:

$$R = (((t, s, Q, p)|t = t_0), U(t_0), ((t, s, Q, p)|t = t_1), U(t_1), \ldots, U(t_{l-1}), ((t, s, Q, p)|t = t_l)),$$

where $head(R) = (t_l, s, Q, p)$ and $t_0, t_1, \ldots, t_l$ are the values of t at each iteration from the last one in which S became nonzero through the iteration in which the call to PACK occurs. The sets $U(t_i)$, $i = 0, \ldots, l$, are the sets Z_{BRK} at step 8(b) of the main algorithm at each value t_i of t. For $i = 0, \ldots, l$, $U(t_i) \neq \emptyset$. However, it may be that $((t, s, Q, p)|t = t_i)$ is void for some values of i. The set $\rho = \{t_i|((t, s, Q, p)|t = t_i)$ is nonvoid$\}$ is the set of *release times in R*. It is also easy to see for any release time t_i that the set $Z(t_i)$, the value of Z at step 10 of the iteration of the main algorithm when $t = t_i$, satisfies $Z(t_i) \subseteq Z(t_l) \cup \bigcup_{j=i+1}^{l} U(t_j)$. Which of the processors in the superset just shown were actually free in $[t_i, t_{j'})$, where j' is the least j satisfying $t_j > t_i$ and $t_j \in \rho$, can be determined by examining A. Notice that if $[t_i, t_{j'}) = [t_i, t_{i+1})$, then we are guaranteed that any $h \in Z(t_i)$ is free for the entire interval $[t_i, t_{j'})$ because the interval corresponds to one iteration of the loop at step 7 of the main algorithm. This property may not hold, however, for interval $[t_i, t_{j'})$ when $t_{i+1} \notin \rho$. In this case we have the following lemma.

LEMMA 4.1. *Let $t_i, t_{j'} \in \rho$, where j' is the least j satisfying $t_j > t_i$, and let $[t_i, t_{i+1})$, $[t_{i+1}, t_{i+2}), \ldots, [t_{j'-1}, t_{j'})$ be the intervals corresponding to the iterations of FAST_CRITICAL_WT from $t = t_i$ to $t = t_{j'-1}$. For $k = i, \ldots, j' - 1$, if $h \in Z(t_{k+1})$, then $h \in Z(t_k)$.*

PROOF. Let some processors become free (be put into Z) at some t_k for $i \leq k < j'$. This event occurs in CLOSE where the processors freed are pushed onto the list Z. By assumption, in the next iteration no jobs are put into N_{NEW}. Thus no processors are freed in SPLIT, and for every processor freed by CLOSE at t_k there is an element in C_a when step 9 is reached at t_{k+1}. Therefore each processor pushed onto Z when $t = t_k$ is reused in the interval $[t_k, t_{k+1})$. □

The procedure PACK employs a rule similar to the one used by Sahni [14]. The rule is applied successively to each interval of the schedule already constructed which begins at a release time and ends with t, the schedule time at which $S = 0$, triggering the call to PACK. Intervals are processed in reverse order on release times in R, that is, "right to left" in the schedule so far constructed. The jobs released at t_i are scheduled when the interval $[t_i, t)$ is processed.

An assignment A is *regular* in $[t_a, t_b)$ if $A' = \{([t_1, t_2), \cdot)|t_2 > t_a\} \subseteq A$ has the property that $A'_i = \{([t_1, t_2), \cdot), ([t_2, t_3), \cdot), \ldots, ([t_{l-1}, t_l), \cdot)\}$ and $t_l \geq t_b$ for $i = 1, \ldots, m$. Figure 4 depicts a regular assignment which for convenience of exposition we show in ascending order on the amount of idle time. In the interval over which regularity is defined, idle time always originates at t_a and is contiguous on any one processor. By Lemma 4.1, the schedule in the first interval to be processed by PACK is regular, and this property is inherited by the schedules of preceding intervals by virtue of properties assured by Lemma 4.1 and the scheduling rule.

Let us assume that the rule of procedure PACK is applied to a regular assignment such as that shown in Figure 4. This rule schedules a very short job at the right of the shortest interval of idle time. A job too large to fit in the shortest interval of idle time is placed on the processor with the largest interval of idle time it can completely fill, the remainder being put as late as possible on the next processor in order of increasing idle time. These alternative placements are illustrated in Figures 5 and 6. In the event a complete interval of idle time is filled, the portions of the lists for interval $[t_a, t_b)$ are swapped, if possible, so that an "uncovered" element in the interval ending with t_a is "covered" on the right. The result of swapping the schedule in Figure 6 is shown in Figure 7. This swapping ensures that the next interval processed will be regular. Swapping is also done to recover preemptions.

In the example of Figure 2, S becomes nonzero in the execution of Algorithm

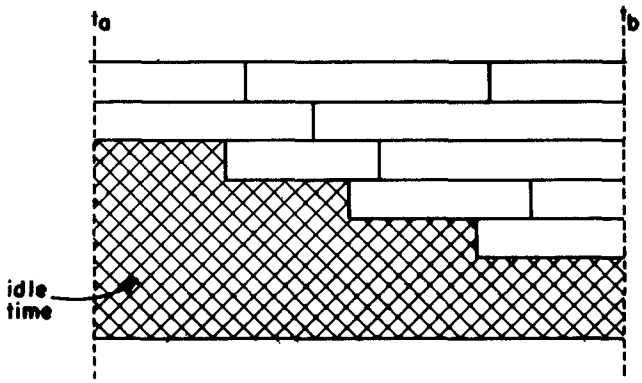

FIG. 4 Example of regularity in $[t_a, t_b)$

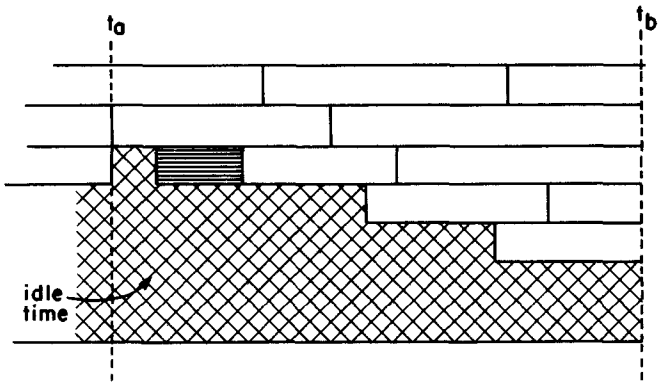

FIG 5 PACK schedules a short job

FAST_CRITICAL_WT at $t = 15$. The one call to PACK occurs when S again becomes zero at $t = 100$. Figure 8 shows A at the point when PACK is called and again when the schedule is completed using the rule just discussed. In this example, the schedule produced has 4 preemptions, none of which is easily recovered, compared to 12 under CRITICAL_WT, 9 of which were recoverable at a cost of $O(nm)$ time. The effect of swapping lists in PACK may be noticed in the changes in processor for tasks T_9, T_{14}, and T_{16}. Tasks scheduled by PACK are shown lightly shaded in the figure.

THEOREM 4.1. *Algorithm FAST_CRITICAL_WT generates schedules which minimize finish time and contain at most $n - 2$ preemptions for $m \geq 2$. This bound on preemptions is a best bound.*

PROOF. Correctness and optimality of the schedules produced follow directly from the correctness of a realization of the procedure PACK and arguments presented earlier. This realization and its correctness proof are given in Appendix D.

If the execution of PACK is ignored, FAST_CRITICAL_WT introduces at most one preemption per task, which occurs when a task becomes noncritical. The execution of PACK introduces at most one preemption of a task, but when it does, the initial task of the job so scheduled begins execution at the time t at which it became noncritical during execution of the loop of the main algorithm. The back pointers into elements preceding t (which are on the lists for A defined in Appendix D) allow the first preemption to be recovered by swapping the parts of the lists which begin at or after t (the B-lists). The details may be seen in the procedure PACK. Thus, ignoring execution of PACK, at most one preemption occurs for any one task.

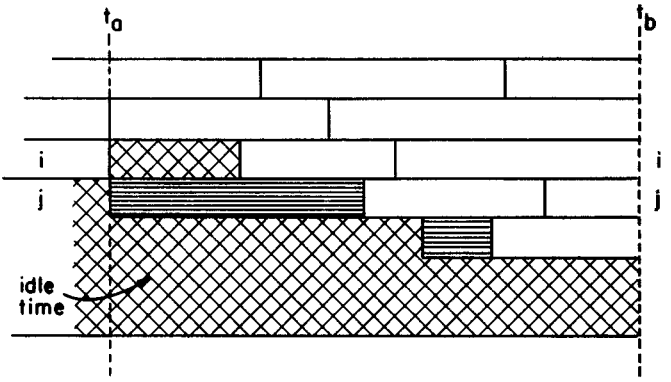

FIG. 6. PACK schedules a job too long to fit in the shortest interval.

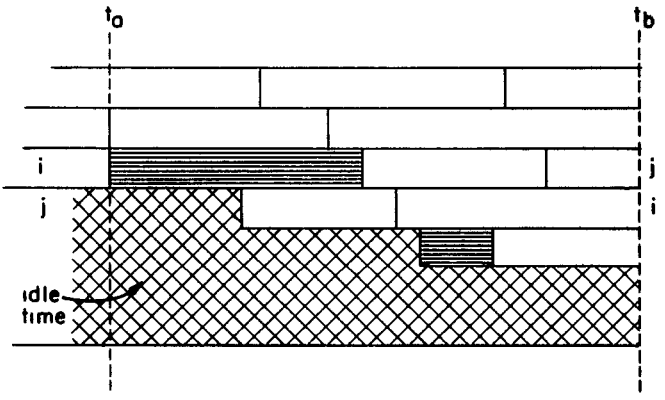

FIG 7 Result of swapping processors in Figure 6.

To obtain the bound $n - 2$, three cases are considered. If no jobs of the given problem are critical, then the algorithm reduces to one execution of PACK in which at least two tasks will receive no preemptions. In the case where there are critical jobs, let there be fewer than two time intervals terminated by the termination of a critical task. If there are two or more such intervals, then at least two tasks are left unpreempted. In the case where there are no time intervals terminated by a critical task, two tasks remain unpreempted in the execution of PACK. If one time interval is terminated by a critical task, then one of its successors will also remain unpreempted.

The case where no job of the given problem is critical establishes $n - 2$ as a best bound on the number of preemptions. □

THEOREM 4.2. *Algorithm FAST_SCHED_BY_WT runs in $O(n \log m)$ time.*

PROOF. Earlier discussion has reduced this proof to a proof that steps 10 and 14 run in $O(n \log m)$ time overall. Over the entire execution of step 10, an element appears in N_{NEW} at most once for each task. Thus step 10 is $O(n)$.

We have already argued a bound of $n - 2$ on the number of preemptions introduced. Consider the execution of PACK (Appendix D). In steps 2, 9, 16, and 18 of PACK each operation is chargeable to some task termination. No individual operation is charged to the same task twice. If Z is kept as a height-balanced search tree [1], then step 12 costs $O(\log m)$, and over the entire execution of the algorithm steps 11 and 12 cost $O(n \log m)$. Steps 13–15 are linear in the number of tasks. Steps unmentioned in this discussion are each constant and add up to $O(n)$. □

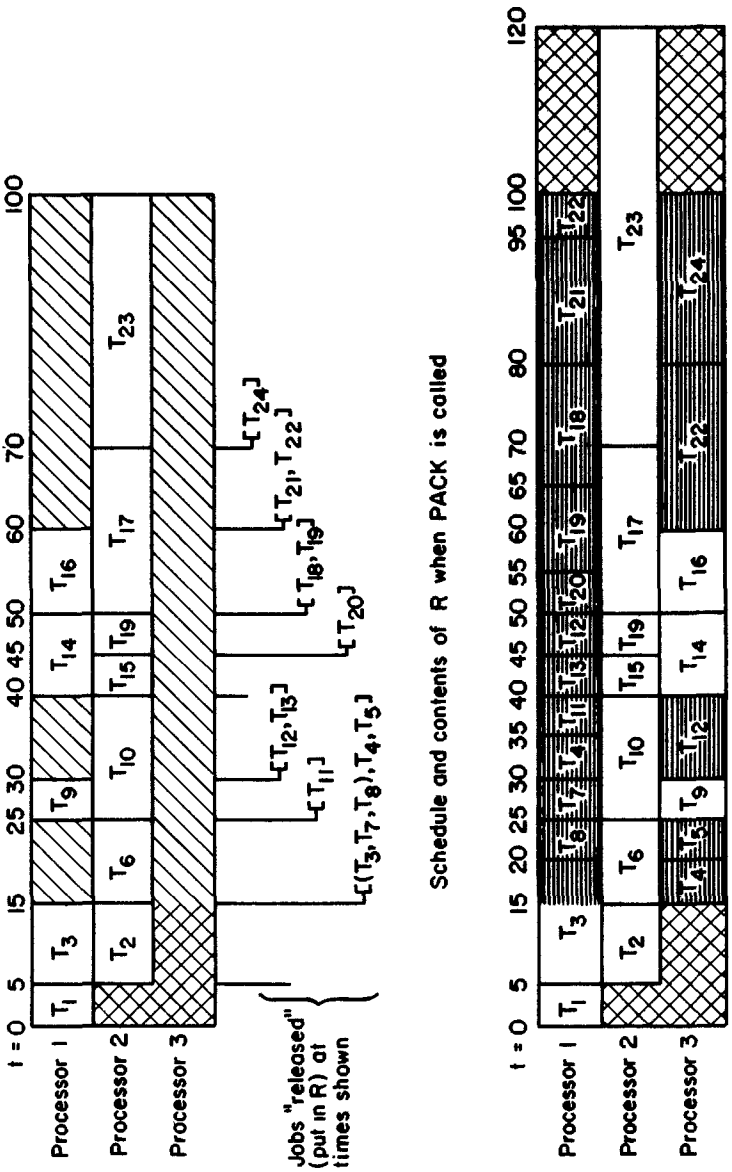

FIG 8. Operation of Algorithm FAST_CRITICAL_WT on three processors for the problem in Figure 2

5. *Conclusion*

The major result of this paper is an $O(n \log m)$ algorithm for scheduling forests composed of n tasks on m identical processors. The schedules produced are optimal with respect to schedule length, and in the worst case the schedules have no more than $n - 2$ preemptions, a bound which cannot be improved when m is large. The schedules produced are a set of m lists, one for each processor, giving in order of execution the tasks to be executed on a given processor.

Also presented is a simpler version of the algorithm which runs in $O(nm)$ time, yielding optimal schedules with no more than $2nm - 4n - m + 3$ preemptions for $m \geq 2$. When $m = 2$, this expression reduces to 1. This algorithm is easily adapted to scheduling independent tasks with release times on-line. The algorithm for the release time problem also runs in $O(nm)$ time, generating schedules with at most $2nm - 2n - m + 2$ preemptions. Virtually the same algorithm also schedules arbitrary forests on two *uniform processors*, processors with uniformly different processing speeds. The only modification is to use the bound of Liu and Yang [10] and the algorithm of Gonzalez and Sahni [5] to schedule the noncritical jobs.

Appendix A. Realization of Procedure RECTANGLE (See Section 2)

```
procedure RECTANGLE(A, N, M, t0, tc),
    procedure SCHED((s, Q), Aj, t0, tc, tf);
        //Given the flattened list (s, Q), procedure SCHED assigns tasks in the interval [t0, min(tc, tf)) Any
        tasks or portions thereof which do not fit in the interval are returned in a flattened list.//
        wlog let Q = ((Δ1, T1), . . , (Δk, Tk));
        i ← 1,
        t ← t0,
        //Schedule task Ti if it will terminate no later than min(tc, tf) //
        while t + Δi ≤ min{tc, tf} do
            push(([t, t + Δi), Ti), Aj),
            t ← t + Δi,
            i ← i + 1,
            if i > k then return ((0, ∅), Aj) endif
        endwhile
        if t = min{tc, tf} then return ((s − t + t0, ((Δi, Ti), . , (Δk, Tk))), Aj)
        else push(([t, min{tc, tf}), Ti), Aj),
            return ((s − min{tc, tf} + t0, ((Δi − min{tc, tf} + t, Ti), (Δi+1, Ti+1), . . . , (Δk, Tk))), Aj)
        endif
    end SCHED;
    //Given a set of tasks (represented by flattened lists in N), the procedure will schedule them on the set of
    processors M from time t0 to time tc If some tasks need to be scheduled after tc, they will be saved in at most
    2k − 1 flattened lists which will be returned in N' //
    wlog let M = {1, . , k},
    N' ← ∅;
    tf ← t0 + ∑(s | (s, Q) ∈ N)/k,
    (s, Q) ← pop(N);
    t ← t0,
    for i ← 1 until k do
        s' ← 0,
        Q' ← ∅,
        while t + s ≤ tf do
            ((s'', Q''), Ai) ← SCHED((s, Q), Ai, t, tc, tf);
                                { (s' + s'', ((Δ'1, T'1),     , (Δ'k', T'k'), (Δ''1, T''1),     , (Δ''k'', T''k'')))
                                    if T'k' ≠ T''1
(s', Q') ←                      { (s' + s'', ((Δ'1, T'1),     , (Δ'k' + Δ''1, T'k'),     , (Δ''k'', T''k'')))
                                    otherwise where wlog
                                        Q' = ((Δ'1, T'1),     , (Δ'k', T'k'))
                                        Q'' = ((Δ''1, T''1),     , (Δ''k'', T''k'')),
            t ← t + s;
            (s, Q) ← pop(N)
        endwhile
        push((s', Q'), N'),
```

```
        if t < t_f then
            ((s', Q'), A_{i+1}) ← SCHED((s, Q), A_{i+1}, t_0, t_c, t_0 + s − t_f + t);
            if t < t_c then ((s', Q'), A_i) ← SCHED((s', Q'), A_i, t, t_c, t_f) endif
            push((s', Q'), N');
            t ← t_0 + s − t_f + t;
            (s, Q) ← pop(N)
        else t ← t_0
        endif
    endfor
    return (A, N')
end RECTANGLE;
```

Appendix B. Proof of Lemma 3.1

PROOF. Prior to the first iteration of the loop at step 7 the partition $(\emptyset, \mathbf{P})$ satisfies part (a) of H1 because $S = 0$, $C \cup N = \emptyset$, and L is a list for $\mathbf{P}$ at $t = 0$. Part (b) is trivially satisfied by the construction of L.

The proof of invariance is by induction. We first prove that part (a) holds after every iteration. Then, with this result, we prove that part (b) holds as well.

Assume (a) of H1 is satisfied before iteration $k \geq 1$. We will show that (a) of H1 is preserved over iteration k. In order to develop the argument, we subscript program variables with statement numbers to stand for the value of the variable *before* the statement is executed in iteration k. For example, before statement 8 the value of set L is denoted L_8. By the assumed correctness of SPLIT, H2 holds before step 9. Let $(\mathbf{P}'_9, \mathbf{P}''_9)$ be the consistent decomposition of $\mathbf{P}$ which satisfies H2. Since the jobs in C are disjoint from the jobs in N, $\mathbf{P}''_9$ has a consistent decomposition $(\mathbf{P}_{C_9}, \mathbf{P}_{N_9})$ where C_9 is a list for $\mathbf{P}_{C_9}$ at t and N_9 is a list for $\mathbf{P}_{N_9}$. Consequently, $(\mathbf{P}'_9 \cup \mathbf{P}_{C_9}, \mathbf{P}_{N_9})$ is a consistent decomposition of $\mathbf{P}$ at t. Let step 9 be executed. The time increment Δ is sufficiently small so that all tasks scheduled in step 9 are independent with respect to $\prec$. Thus $\mathbf{P}'_9 \cup \mathbf{P}_{C_9}$ has a consistent decomposition $(\mathbf{P}_{A_{10}}, \mathbf{P}_{C_{10}})$ where A_{10} is a complete feasible assignment of $\mathbf{M}$ to $\mathbf{P}_{A_{10}}$ in $[0, t + \Delta)$ and C_{10} is a list for $\mathbf{P}_{C_{10}}$ at $t + \Delta$. It should be noted that $\mathbf{P}_{C_{10}}$ is a pseudoproblem in the sense that there may be $(s, u, T) \in \mathbf{P}_{C_{10}}$ for which $u - (t + \Delta) = 0$. Since $N_9 = N_{10}$, it follows that $(\mathbf{P}_{A_{10}}, \mathbf{P}_{C_{10}} \cup \mathbf{P}_{N_{10}})$ is a consistent decomposition of $\mathbf{P}$ where C_{10} is a list for $\mathbf{P}_{C_{10}}$ at $t + \Delta$ and N_{10} is a list for $\mathbf{P}_{N_{10}}$.

Consider the execution of steps 10 and 11. From Lemma 2.1, the disjointness of C and N, and the fact that $C_{10} = C_{12}$, it can be concluded that $(\mathbf{P}_{A_{12}}, \mathbf{P}_{C_{12}} \cup \mathbf{P}_{N_{12}})$ is a consistent decomposition for $\mathbf{P}$ where A_{12} is a complete feasible assignment of $\mathbf{M}$ to $\mathbf{P}_{A_{12}}$ in $[0, t + \Delta)$, C_{12} is a list for $\mathbf{P}_{C_{12}}$ at $t + \Delta$, and N_{12} is a list for $\mathbf{P}_{N_{12}}$. It remains to observe that step 12 replaces C_{12} with the sets C_{13} and L_{13} having the property that $(\mathbf{P}_{C_0}, \mathbf{P}_{C_{13}} \cup \mathbf{P}_{L_{13}})$ is a consistent decomposition for $\mathbf{P}_{C_{12}}$ where $\mathbf{P}_{C_0}$ is a pseudoscheduling problem for which τ is identically zero, C_{13} is a list for $\mathbf{P}_{C_{13}}$ at $t + \Delta$, and L_{13} is a list for $\mathbf{P}_{L_{13}}$ at $t + \Delta$. Combining results and noticing that $A_{13} = A_{12}$ and $N_{13} = N_{12}$ give a consistent decomposition $(\mathbf{P}_{A_{13}}, \mathbf{P}_{C_{13}} \cup \mathbf{P}_{L_{13}} \cup \mathbf{P}_{N_{13}})$ of $\mathbf{P}$, which satisfies (i) and (ii) of part (a) of H1 at the start of iteration $k + 1$. The effect of step 11 satisfies (iii). Induction on k completes the proof that part (a) of H1 is invariant.

In order to prove the invariance of part (b) of H1, we must prove the following assertion: At any iteration $k \geq 1$, $N_8 \subseteq N_9$ and $C_9 \subseteq C_8 \cup L_8$. The proof is by contradiction. Assume there exists $(s, ((\Delta_1, T_1), \ldots)) \in N_8$ and that $(s + t, \Delta_1 + t, T_1) \in C_9$ in satisfaction of H2. Since $N_8 \neq \emptyset$, it must be that $k > 1$ and $card(C_8 \cup L_8 \cup N_8) \geq m$. The assumed contradictory element must have been a member of N_{12} in iteration $k - 1$.

In proving part (a) of H1 we identified a consistent decomposition $(\mathbf{P}_{A_{12}}, \mathbf{P}_{C_{12}} \cup \mathbf{P}_{N_{12}})$ of $\mathbf{P}$ prior to the execution of step 12. Let this decomposition occur in iteration $k - 1$. If we take $C_{12} \cup N_{12}$ to be a list for $\mathbf{P}_{C_{12}} \cup \mathbf{P}_{N_{12}}$ at $t_k = t_{k-1} + \Delta_{k-1}$, where t_i is the value of program variable t at the start of iteration i, then without loss of generality we can write $S_1 \geq \cdots \geq S_{j^*} \geq S_l \geq \cdots \geq S_r$ if $\mathbf{P}_{C_{12}} \cup \mathbf{P}_{N_{12}} = \bigcup_{i=1}^{r} \mathbf{J}_i$. Here j^* is the critical index of iteration $k - 1$. Therefore $(m - j^*)S_{j^*} > \sum_{i=l}^{r} S_i$, but $(m - l)S_l \leq \sum_{i=l+1}^{r} S_i$. Now consider

the scheduling problem $\mathbf{P}'_8$ of the consistent decomposition $(\mathbf{P}'_8, \mathbf{P}''_8)$ which satisfies part (a) of H1 before step 8 in iteration k. $\mathbf{P}''_8$ is derived from $\mathbf{P}_{C_{12}} \cup \mathbf{P}_{N_{12}}$ of iteration $k-1$ by the execution of step 12. Without loss of generality let $\mathbf{P}''_8 = \sum_{i=1}^{r''} \mathbf{J}''_i$ satisfy $S''_1 \geq \cdots \geq S''_{l''} \geq \cdots \geq S''_{r''}$, where $\mathbf{J}_l = \mathbf{J}''_{l''}$ and thus $S_l = S''_{l''}$. We may assume that $\mathbf{J}''_{l''}$ is a job which contradicts $N_8 \subseteq N_9$ and $C_9 \subseteq C_9 \cup L_8$. In this case the critical index in iteration k is greater than or equal to l''. Steps 12 and 13 in iteration $k-1$ had no effect on jobs $\mathbf{J}_l, \ldots, \mathbf{J}_r$, so $\{\mathbf{J}_{l+1}, \ldots, \mathbf{J}_r\} \subseteq \{\mathbf{J}''_{l''+1}, \ldots, \mathbf{J}''_{r''}\}$. Consequently $\sum_{i=l''+1}^{r''} S''_i \geq \sum_{i=l+1}^{r} S_i$. Our assumption is that

$$(m - l'')S''_{l''} > \sum_{i=l''+1}^{r''} S''_i.$$

We consider two cases. Let l'' satisfy $l \leq l'' \leq m$. Our assumption clearly fails if $l'' = m$. Therefore $l'' < m$ and

$$S''_{l''} = S_l \leq \sum_{i=l+1}^{r} \frac{S_i}{(m-l)} \leq \sum_{i=l''+1}^{r''} \frac{S_i}{(m-l'')},$$

which contradicts our assumption. Thus the other case, $l'' < l \leq m$, is the only possibility. In this case at least $(l - l'')$ jobs from the set $\{\mathbf{J}_1, \ldots, \mathbf{J}_r\}$ have been split, so their weights appear in the sequence $S''_{l+1}, \ldots, S''_{r''}$. Thus

$$\sum_{i=l''+1}^{r''} S''_i > \sum_{i=l+1}^{r} S_i + (l - l'')S_l \geq (m-l)S_l + (l-l'')S_l = (m-l'')S_l$$

and

$$S_l = S''_{l''} > \sum_{i=l''+1}^{r''} \frac{S''_i}{(m-l'')} > \frac{(m-l'')S_l}{(m-l'')} = S_l,$$

a contradiction. All cases have been exhausted, so we conclude that $N_8 \subseteq N_9$ and $C_9 \subseteq C_8 \cup L_8$ over the execution of SPLIT, as claimed. It remains to notice that on all iterations but the first, every job represented in $C_8 \cup L_8$ has a direct predecessor in set C_9 of the previous iteration, and part (b) of H1 follows by induction on k. □

Appendix C. Realization of Procedures SPLIT and CLOSE (See Section 4)

```
procedure SPLIT(C_a, C_d, L, S, t, Z, Z_BRK)
    N ← ∅;
    for (s_i, u_i, T_i, p_i) ∈ L do
        //Move a job from L to C_a//
        C_a ← C_a ∪ {(s_i, u_i, T_i, p_i)},
        let (s, u, T, p) minimize s in C_a ∪ C_d,
        //Keep only critical jobs in C_a ∪ C_d//
        while card(C_a ∪ C_d) ≥ m or (S > 0 and S ≥ (s − t)(m − card(C_a ∪ C_d))
            and card(C_a ∪ C_d) > 0) do
            if (s, u, T, p) ∈ C_a then C_a ← C_a − {(s, u, T, p)}
            else //Complete schedule entry and record freeing of processor//
                C_d ← C_d − {(s, u, T, p)};
                let head(A_k) = elem(p),
                ([t_1, ∞), T) ← pop(A_k);
                push([t_1, t), T), A_k),
                push(k, Z),
                Z_BRK ← Z_BRK ∪ {k}
            endif
            //Put job deleted from C_a ∪ C_d into N and update S//
            N ← N ∪ {(s − t, ((u − t, T), (τ(T_1), T_1), ..., (τ(T_k), T_k)), p) | where wlog T ≺ T_i iff 1 ≤ i ≤ k and
                    for 1 ≤ i, j ≤ k if T_i ≺ T_j then i < j},
            S ← S + s − t;
```

```
            if card(C_a ∪ C_d) > 0 then let (s, u, T, p) minimize s in C_a ∪ C_d endif
        endwhile
    endfor
    if card(C_a ∪ C_d) = 0 then Δ ← S/m
    else Δ ← min({u − t | (s, u, T, p) ∈ C_a ∪ C_d} ∪ {S/(m − card(C_a ∪ C_d))})
    endif
    return (C_a, C_d, ∅, N, S, Δ, Z, Z_BRK)
end SPLIT;

procedure CLOSE(C_d, t, Δ, Z_BRK);
    L ← ∅;
    //For each job in C_d for which the initial task has zero execution time remaining, terminate this task in the
    schedule and put successor jobs in L//
    while C_d ≠ ∅ do
        let (s, u, T, p) minimize u in C_d,
        if u − (t + Δ) > 0 then return (L, C_d, Z, Z_BRK) endif
        C_d ← C_d − {(s, u, T, p)};
        let head(A_k) = elem(p);
        ([t_1, ∞), T_i) ← pop(A_k);
        push(([t_1, t + Δ), T_i), A_k),
        push(k, Z),
        Z_BRK ← Z_BRK ∪ {k},
        for T_j satisfying T ≺ T_j and, for no T_l, T ≺ T_l ≺ T_j do
            L ← L ∪ {(σ(T_j) + t + Δ, τ(T_j) + t + Δ, T_j, Λ)}
        endfor
    endwhile
    return (L, C_d, Z, Z_BRK)
end CLOSE;
```

Appendix D. Realization of Procedure PACK and Proof of Correctness (see Section 4)

```
procedure PACK(A, R, Z, t),
    procedure SWAP(x);
        //Swap list B_w with list B_j, for j ∈ Z_SWAP//
        w ← x;
        if t_1 < t_R where head(A_w) = ([·, t_1), ·) then
            let j ∈ Z_SWAP,
            B_w ↔ B_j;
            Z_SWAP ← Z_SWAP − {j};
            w ← j
        endif
        if p_R ≠ Λ then
            let elem(p_R) = head(A_k),
            B_w ↔ B_k;
            if k ∈ Z_SWAP then Z_SWAP ← (Z_SWAP ∪ {w}) − {k} endif
        endif
    end SWAP;
    procedure SCHED(Q, B_i, t_1, t_2, B_j, t_3, t_4)
        //Schedule Q so as to fill idle time on B_i, if possible, with any overflow scheduled as late as possible on
        B_j//
        wlog let Q = ((Δ_1, T_1), . . , (Δ_l, T_l)),
        L ← ∅;
        t ← t_1,
        k ← 1;
        while t + Δ_k ≤ t_2 do
            push(([t, t + Δ_k), T_k), L);
            t ← t + Δ_k;
            k ← k + 1
        endwhile
        if t < t_2 then
            push(([t, t_2), T_k), L),
            while L ≠ ∅ do push(pop(L), B_i) endwhile
            t ← t_3 + Δ_k − (t_2 − t);
            push(([t_3, t), T_k), L),
            k ← k + 1
```

```
        else while L ≠ ∅ do push(pop(L), B_i) endwhile
            t ← t_3
        endif
        while k ≤ l do
            push(([t, t + Δ_k), T_k), L),
            t ← t + Δ_k,
            k ← k + 1
        endwhile
        while L ≠ ∅ do push(pop(L), B_j) endwhile
        return (B_i, B_j)
    end SCHED;
    //Accumulate Z_BRK sets at the head of R//
1.  Y ← ∅
2   while head(R) ⊊ {1, . . , m} do Y ← Y ∪ pop(R) endwhile
3.  if R = ∅ then return (A, ∅, Z) endif
    //Move prefix of each list A_i, up to any idle time beginning at last release time, onto a new list B_i//
4.  Z_SWAP ← ∅,
5   (t_R, s_R, Q_R, p_R) ← head(R),
6   for i ∈ Z do  B_i ← {([t, t), Λ)};
                  while t_R < t_1 = t_2 where head(A_i) = ([·, t_1), ·), head(B_i) = ([t_2, ·), ·) do push(pop(A_i), B_i)
                  endwhile
                  if t_1 = t_R < t_2 where head(A_i) = ([·, t_1), ·), head(B_i) = ([t_2, ·), ·) then Z_SWAP ← Z_SWAP ∪ {i}
                  endif
    endfor
7   while R ≠ ∅ do
        case
8       head(R) = (·, ·, ·, ·): //Move prefix of each list A_i, where i was a member of Z_BRK, up to first gap in
                                A_i//
9                               [(t_R, s_R, Q_R, p_R) ← pop(R);
                                for i ∈ Y do
                                    if i ∉ Z then B_i ← {([t, t), Λ)},
                                        Z ← Z ∪ {i} endif
                                    while t_R < t_1 = t_2 where head(A_i) = ([·, t_1), ·), head(B_i) = ([t_2, ·), ·) do
                                        push(pop(A_i), B_i)
                                    endwhile
                                    if t_1 = t_R < t_2 where head(A_i) = ([·, t_1), ·), head(B_i) = ([t_2, ·), ·) then
                                        Z_SWAP ← Z_SWAP ∪ {i}
                                    else Z_SWAP ← Z_SWAP − {i}
                                    endif
                                endfor
10.                             Y ← ∅;
                                //Schedule job (t_R, s_R, Q_R, p_R)//
11                              wlog let Z = {1, . , q}, and
                                let π(Z) satisfy
                                    (i) for i = 1, , q, π(i) ∈ Z
                                    (ii) for i = 1, . . , q − 1, t_π(i) ≤ t_π(i+1),
                                        head(B_π(i)) = ([t_π(i), ·), ·),
                                        head(B_π(i+1)) = ([t_π(i+1), ·), ·)},
12                              l ← min{l | t_π(l) − t_R ≥ s_R where 1 ≤ l ≤ q and head(B_π(l)) = ([t_π(l), ·), ·)};
                                case
13                                  t_π(l) − t_R = s_R [(B_π(l), ·) ← SCHED(Q_R, B_π(l), t_R, t_π(l), ·, ·, ·), SWAP(l)]
14                                  l = 1 or t_π(l−1) = t_R· [(B_π(l), ·) ← SCHED(Q_R, B_π(l), t_π(l) − s_R, t_π(l), ·, ·, ·)]
15                                  else. [(B_π(l−1), B_π(l)) ← SCHED(Q_R, B_π(l−1), t_R, t_π(l−1), B_π(l),
                                        t_π(l) − s_R + t_π(l−1) − t_R, t_π(l)),
                                        SWAP(l − 1)]
                                endcase
16      head(R) ⊊ {1, , m}. //Head of R is a processor set//
                                [Y ← Y ∪ pop(R)]
        endcase
    endwhile
    //Move elements from B-lists back to A-lists//
17. X ← ∅;
18. for i ∈ Z do
        while head(B_i) ≠ ([t, t), ·) do
            ([t_1, t_2), T_j) ← pop(B_i),
```

```
            if t1 = t4 and Tj = Tk where head(Ai) = ([t3, t4), Tk) then
                ([t3, t4), Tk) ← pop(Ai);
                push(([t3, t2), Tk), Ai)
            else push(([t1, t2), Tj), Ai)
            endif
        endwhile
        if t2 = t then X ← X ∪ {i} endif
    endfor
19. return (A, R, X)
end PACK;
```

The proof of correctness of procedure PACK will be by induction over the execution of the loop at step 7 of FAST_CRITICAL_WT. It will be necessary to reference the properties of Algorithm CRITICAL_WT established in earlier sections. Let CRITICAL_WT(**P**, m) = $(A, \cdot)$. For any value t_k assumed by t during the execution of CRITICAL_WT(**P**, m), let $A(t_k) = \{([t_1, t_2), \cdot) | t_2 \le t_k\} \subseteq A$, and let $\mathbf{P}(t_k)$ be the scheduling problem for which $A(t_k)$ is a complete feasible assignment of **M** to $\mathbf{P}(t_k)$ in $[0, t_k)$. Also let A^t be any assignment A in which each instance of ∞ is replaced by t. Referring to FAST_CRITICAL_WT, let $R(t_k) = \{(t, s, Q, p) | (s, Q, p) \in N_{\text{NEW}}(t_k)\}$ be the jobs put into R when $t = t_k$, and let $S(t_k)$ be the value of S before step 8 when $t = t_k$. $S(t_k)$ is the total weight of all jobs which become noncritical before t_k and which CRITICAL_WT schedules after t_k. Now consider the execution of PACK.

Define

$$Z_{t_R} = \{i | ([t_1, t_2), \cdot) \in A_i \cup B_i \text{ implies } t_2 \le t_R \text{ or } t_R \le t_1\},$$
$$R_{t_R} = \{(t, s, Q, p) | t = t_R\} \subseteq R,$$
$$S(R_{t_R}) = \sum(s | (t, s, Q, p) \in R_{t_R}),$$

where R, t_R, A_i, B_i are program variables in PACK. The following lemma may be proved by induction over the execution of PACK.

LEMMA 4.2. *$Z_{SWAP} = \{i | ([t_1, t_R), \cdot) \in A_i$ and $t_2 > t_r$ where $([t_2, t_3), \cdot)$ minimizes t_2 in $B_i\}$ after step* 10 *in any iteration of PACK.*

LEMMA 4.3. *Let $R_{t_R} = R(t_R)$ after step* 10 *in some iteration of PACK, and let Z^0_{SWAP} be the value of Z_{SWAP} at this time. Then*

$$(i) \quad \frac{S(R(t_R))}{card(Z^0_{SWAP})} \ge \frac{S(t_R)}{card(Z(t_R) - Z^0_{SWAP})},$$

$$(ii) \quad \frac{S(R(t_R)) - s}{card(Z^0_{SWAP}) - 1} > \frac{S(t_R)}{card(Z(t_R) - Z^0_{SWAP})},$$

for any $(t, s, Q, p) \in R(t_R)$.

PROOF. Part (i) is a consequence of the definition of job criticality implemented in CRITICAL_WT. Part (ii) follows from the same arguments since s must be less than or equal to the sum of the weights of all of the descendants of Q's predecessor job with respect to $\prec$. □

Let G be the following proposition:

(a) $\hat{A} = \cup_{i=1}^{m}(A_i \cup B_i)$ is regular in $[t_R, t)$;
(b) $([t_1, t_2), \cdot) \in A_i$ iff $t_2 \le t_R$ for $i \in Z$;
(c) $Z_{t_R} \subseteq Z$;
(d) there exists A^*, a complete feasible assignment of **M** to $\mathbf{P}^* = \cup_{i=1}^{r} \{\mathbf{J}_i^* | (t_i, s_i, Q_i, p_i) \in R$, Q_i is a list for $\mathbf{J}_i^*$ and $\prec_i^*$ is a total order$\}$ in $[t_0, t)$ and $A^* \cup \hat{A}^t$ is a complete feasible assignment of **M** to $\mathbf{P}(t)$ in $[t_0, t)$.

LEMMA 4.4. *G is invariant over the execution of PACK when G is evaluated after step* 10.

PROOF. Consider the first execution of step 10. Steps 1–10 establish parts (a), (b), and (c) of G. Part (d) follows from the correctness of Algorithm CRITICAL_WT. Let G hold following some execution of step 10 and consider the next execution of step 10. There are two cases.

Case 1. *The variable t_R is unchanged.* In this case exactly one iteration of the loop at step 7 is executed. Part (b) is preserved by step 9. A trace of the remaining steps of the loop will verify that parts (a) and (c) are preserved. It remains to show that part (d) will hold when a job has been scheduled in the manner specified in steps 11–15. Because the assignment in $\hat{A}$ is regular in $[t_R, t)$, the idle time on any processor i after time t_R coincides with an interval $[t_R, t_i)$ where t_i is the first time to show on the list B_i. Every processor with idle time in $[t_R, t)$ appears in the set Z as a consequence of part (c). Clearly the scheduling rule in PACK preserves regularity, as was shown in Figures 5 and 6. Because the precedence relations in each job in R are a total order, the jobs in R can be viewed as single tasks. Consequently there is an interchange argument to show that A^* assumed by G can be constructed so that the job Q_R is scheduled as in PACK. Thus part (d) is preserved.

Case 2. *The variable t_R assumes a new value.* In this case one or more executions of step 16 intervene after the iteration in which G is given as true. By the arguments above, G certainly holds at the end of the iteration in which G is given as true. Now let t_R assume a new (smaller) value. Again, by the arguments in case 1, part (d) must hold. Also, the execution of step 9 reestablishes parts (b) and (c). The issue is whether part (a) (regularity) is preserved.

Lemma 4.1 establishes regularity at all times except at the former t_R. We must show that G true after step 15, when $(t, s, Q, p) \in R$ implies $t < t_R$, itself implies $([t_1, t_R), \cdot) \in A_i$ only if $([t_R, t_2), \cdot) \in B_i$. The set Z_{SWAP} contains all processors which violate this property. Each time the idle time on some processor is used up, a processor is removed from Z_{SWAP}. Our proof proceeds by induction on $card(Z_{\text{SWAP}})$.

It follows by the proof of correctness of CRITICAL_WT that $I_{t_R} = \sum_{i \in Z_{t_R}} (t_1 - t_r | ([t_1, t_2), \cdot)$ minimizes t_1 in $B_i) = S(t_R) + S(R_{t_R})$ after step 10 in any iteration of PACK. Let $card(Z_{\text{SWAP}}) = 1$. Since $I_{t_R} - \min_{i \in Z_{t_R}} \{t_1 - t_R | ([t_1, t_2), \cdot)$ minimizes t_1 in $B_i\} \geq S(t_R)$, PACK will surely remove a processor from Z_{SWAP} while scheduling $R(t_R)$. So let $card(Z_{\text{SWAP}}) = a > 1$, and assume that PACK correctly reduces Z_{SWAP} to $\varnothing$ whenever $card(Z_{\text{SWAP}}) = a - 1$. By the argument just presented for $card(Z_{\text{SWAP}}) = 1$, it must occur that a processor is removed from Z_{SWAP}. Part (i) of Lemma 4.3 is given. If the first processor removed from Z_{SWAP} is the one with minimum idle time, let δ be the length of that interval. By minimality of δ, $a\delta \leq S(R(t_R))$. Thus

$$\frac{S(R(t_R)) - \delta}{a - 1} \geq \frac{S(R(t_R))(1 - (1/a))}{a - 1} = \frac{S(R(t_R))}{a} > \frac{S(t_R)}{card(Z(t_R)) - a},$$

and we can conclude from our induction hypothesis that PACK will correctly reduce Z_{SWAP} to $\varnothing$. If the first processor removed from Z_{SWAP} is not the one with minimum idle time, we can let it be removed in the first iteration. If the weight of the job so scheduled is s, then we have the relation of part (ii) of Lemma 4.3, and the proof is again completed by induction on a. It should be noticed that in each of the arguments just presented, the last job scheduled on the processor with least idle time was assumed to be split so that the job scheduled exactly filled the idle interval. □

ACKNOWLEDGMENT. We wish to acknowledge a referee for calling the paper of Davida and Linton to our attention and suggesting we discuss the relation of this work to ours.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. COFFMAN, E.G. JR., ED. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, New York, 1976.
3. CONWAY, R.W., MAXWELL, W.L., AND MILLER, L.W. *Theory of Scheduling* Addison-Wesley, Reading, Mass., 1967.
4. DAVIDA, G.I., AND LINTON, D.J. A new algorithm for the scheduling of tree structured tasks. Proc. 1976 Conf. Inform. Sci. and Syst., Baltimore, Md., 1976, pp. 543–548.
5 GONZALEZ, T., AND SAHNI, S Preemptive scheduling of uniform processor systems. *J. ACM 25,* 1 (Jan. 1978), 92–101.
6. HORN, W.A Some simple scheduling algorithms. *Naval Res. Log. Quart. 21* (1974), 177–185
7 HORVATH, E.C., LAM, S., AND SETHI, R. A level algorithm for preemptive scheduling. *J. ACM 24,* 1 (Jan 1977), 32–43.
8. HU, T.C Parallel sequencing and assembly line problems. *Operations Res. 9,* 6 (Nov. 1961), 841–848.
9. LAM, S , AND SETHI, R Worst case analysis of two scheduling algorithms. *SIAM J. Comptg. 6* (1977), 518–536
10 LIU, J W.S., AND YANG, A. Optimal scheduling of independent tasks on heterogeneous computing systems. Proc. 1974 ACM Annual Conf , 1974, San Diego, Calif., pp. 38–45.
11. MCNAUGHTON, R. Scheduling with deadlines and loss functions. *Management Sci. 12,* 7(1959), 1–10.
12. MUNTZ, R.R., AND COFFMAN, E.G. JR Optimal preemptive scheduling on two-processor systems. *IEEE Trans. Comptr. C-18,* 11(1969), 1014–1020.
13 MUNTZ, R.R., AND COFFMAN, E G JR Preemptive scheduling of real-time tasks on multiprocessor systems. *J. ACM 17,* 2 (April 1970), 324–338.
14. SAHNI, S. Preemptive scheduling with due dates. *Operations Res. 27,* 5 (Sept.–Oct. 1979), 925–934.

RECEIVED AUGUST 1977; REVISED JUNE 1979; ACCEPTED JUNE 1979