Theoretical Computer Science 12 (1980) 1–17
© North-Holland Publishing Company

EFFICIENT SCHEDULING OF TASKS WITHOUT FULL USE OF PROCESSOR RESOURCES*

Jeffrey M. JAFFE**
Laboratory for Computer Science, MIT, Cambridge, MA 02139, U.S.A.

Communicated by R. Karp
Received December 1978
Revised July 1979

Abstract. The nonpreemptive scheduling of a partially ordered set of tasks on a machine with m processors of different speeds is studied. Polynomial time algorithms are presented which benefit from selective non-use of slow processors. The performance of these heuristics is asymptotic to $\sqrt{m}$ times worse than optimal, whereas list schedules are unboundedly worse than optimal for any fixed value of m. The techniques of analyzing these schedules are used to obtain a bound on a large class of preemptive schedules.

1. Introduction

The problem of scheduling a partially ordered set of tasks on a machine with m processors of different speeds was introduced by Liu and Liu [10, 11]. They studied a class of schedules known as demand driven or list schedules. The characteristic property of these schedules is that at no time is there both an idle processor and an unexecuted executable task. They showed that any list schedule has a finishing time which is at most $1+(b_1/b_m)-(b_1/(b_1+\cdots+b_m))$ times worse than optimal, where b_i is the speed of the ith fastest processor (the optimal schedule is the one with least finishing time). In addition, list schedules do at times perform as poorly as the bound [10, 11]. This is a discouraging result since a large gap between the speeds of the fastest and slowest processors implies the ineffectiveness of list scheduling, independent of the speeds of the other processors or the number of processors. List scheduling has been a prototype of approximation algorithms since its introduction in the identical processor case [4].

One way of avoiding the problem of unboundedly bad behavior is to use preemptive scheduling. When one is allowed to use preemption, one is allowed to temporarily suspend the execution of an executing task, and complete the execution

* This report was prepared with the support of a National Science Foundation graduate fellowship, and National Science Foundation grant No. MCS77-19754. The computations on MACSYMA were supported by ERDA contract NO. E(11-1)-3070 and NASA grant No. NSG1323.

** Current address: IBM, T. J. Watson Research Center, Yorktown Heights, NY10598, U.S.A.

of the task later. Furthermore there is no added cost for this preemption. Horvath, Lam and Sethi studied a 'level algorithm' for the preemptive scheduling of tasks [6] which generalizes the algorithms of [13, 14]. Schedules produced by the level algorithm are special cases of maximal usage schedules [7]. The worst case performance of any maximal usage schedule is $\frac{1}{2}+\sqrt{m}$ times worse than optimal [7]. Any preemptive schedule may be easily transformed into a maximal usage schedule, with a finishing time at least as small as the original schedule.

A related model in which better algorithms are known is the nonpreemptive scheduling of independent tasks (i.e. no partial order) on processors of different speeds. In that case, there are ε-approximation algorithms due to Horowitz and Sahni [5] which require an exponential amount of time as a function of the number of processors, but not as a function of the number of tasks. Gonzalez, Ibarra and Sahni [3] show that the largest processing time heuristic (a polynomial time algorithm) is at most twice optimal. Finally, Cho and Sahni [1] have studied various other algorithms for this machine environment.

The decision problem of determining whether a given set of tasks (even if they are independent) can be scheduled on processors of different speeds within a given finishing time is NP-complete [5]. As a result it is rather unlikely that an algorithm can be found which runs in polynomial time and always produces the optimal schedule. It is for this reason that we try to develop algorithms that approximate the optimal schedule as we proceed to explain.

The focus of this paper is to provide a nonpreemptive algorithm which is guaranteed to be no worse than $O(\sqrt{m})$ times worse than optimal, regardless of the speeds of the processors. While $\sqrt{m}$ and b_1/b_m (the leading behavior in the performance of list schedules) are strictly speaking incomparable (in that either may be smaller for different sets of processor speeds), the natural way that the algorithm is developed guarantees that the worst case performance for any fixed set of processor speeds is not worse than $1+b_1/b_m$.

The basic strategy of the heuristic is to use only the fastest i processors for an appropriately chosen value of i. In contrast to other algorithms that rule out the use of certain processors [3], this algorithm is unique in that the processors may be ruled out before the set of tasks is examined. In particular, the algorithm does not need any information about the time requirement of a task before scheduling the task.

Formal definitions are provided in Section 2. Section 3 describes which processors are to be used if $O(\sqrt{m})$ behavior is desired. A bound of $\sqrt{m}+O(m^{1/4})$ is obtained on the performance of the heuristic. Also, for small values of m the exact worse case performance of the algorithm is computed. This is significant as $O(m^{1/4})$ is potentially a dominating factor for small values of m. The time complexity of the algorithm is $O(n^2+m)$, where n is the number of tasks. It is significant to note that any algorithm which does not use information about the time requirement of a task before scheduling the task, cannot perform better than $\sqrt{m}$ times worse than optimal [2].

As mentioned above, in [7] a bound of $\sqrt{m}+\frac{1}{2}$ times worse than optimal is obtained for the maximal usage preemptive schedules. Section 4 describes the application of the techniques of this paper to that class of preemptive schedules. It is shown that the analysis of Section 3 provides a bound of $\sqrt{m}+O(m^{1/4})$ on these algorithms. While the analysis of [7] provides a better bound, it is worthwhile to note that the correct asymptotic behavior may be derived without using the techniques of [7].

2. Task systems

A *task system* $(\mathcal{T}, <, \mu)$ consists of:

(1) a set $\mathcal{T}$ of n *tasks*,

(2) a *partial ordering* $<$ on $\mathcal{T}$,

(3) a *time function* $\mu : \mathcal{T} \to \mathbf{R}^+$.

The set $\mathcal{T}$ represents the set of tasks or jobs that need to be executed. The partial ordering specifies which tasks must be executed before other tasks. The value $\mu(T)$ is the *time requirement* of the task T.

Associated with the system $(\mathcal{T}, <, \mu)$ is a set of processors $\mathcal{P}=\{P_i: 1\leq i\leq m\}$. There is a *rate* b_i associated with the processor $P_i (b_1 \geq b_2 \geq \cdots \geq b_m > 0)$. If a task T is assigned to a processor P with rate b, then $\mu(T)/b$ time units are required for the processing of T on P. (When discussing a generic processor 'P', the associated rate is taken to be 'b'.)

A *schedule* for $(\mathcal{T}, <, \mu)$ *on a set of processors* $\mathcal{P}$ *with rates* $b_1, \ldots, b_m$ is a total function $\mathcal{S}: \mathcal{T} \to \mathbf{R} \times \mathcal{P}$ satisfying conditions (1) and (2) below. If $\mathcal{S}(T)=(t, P)$, then the *starting time* of task T is t and the *finishing time* of T is $t+(\mu(T)/b)$ and T is *being executed on* P for times x such that $t \leq x < t+(\mu(T)/b)$.

The function $\mathcal{S}$ satisfies:

(1) for all $t \in \mathbf{R}^+$, if two tasks are both being executed at time t, then they are being executed on different processors at time t;

(2) for $T, T' \in \mathcal{T}$, if $T < T'$ the starting time of T' is no less than the finishing time of T.

Condition (1) asserts that processor capabilities may not be exceeded. Condition (2) forces the obedience of precedence constraints.

The *finishing time* of a schedule is the maximum finishing time of the set of tasks. An *optimal schedule* is any schedule that minimizes the finishing time. For two schedules $\mathcal{S}$ and $\mathcal{S}'$, with finishing times w and w' the *performance ratio of* $\mathcal{S}$ *relative to* $\mathcal{S}'$ is w/w'.

One class of schedules is the class of *list schedules*. List schedules are designed to avoid the apparently wasteful behavior of letting a processor be idle while there are executable tasks. A list schedule uses a (priority) *list* L which is a permutation of the tasks of $\mathcal{T}$, i.e., $L=(T_1, \ldots, T_n)(T_i \in \mathcal{T}$ and for $i \neq j$, $T_i \neq T_j)$. The *list schedule* for $(\mathcal{T}, <, \mu)$ with the list L is defined as follows. At each point in time that at least one processor completes a task, each processor that is not still executing, is assigned an unexecuted executable task. The tasks are chosen by giving higher priority to those unexecuted tasks with the lowest indices in L. If r processors are simultaneously available ($r > 1$), then the r highest priority unexecuted, executable tasks are assigned to the available processors. The decision as to which processor gets which task is made arbitrarily (or one may choose to assign higher priority tasks to faster processors). Only if not enough unexecuted tasks are executable, do processors remain idle. Note that any schedule that is unwasteful in the sense that processors are never permitted to be idle unless no free tasks are available can be formulated as a list schedule.

When the processors are of different speeds, list schedules may be as bad as $1+(b_1/b_m)-(b_1/(b_1+\cdots+b_m))$ times worse than optimal [10, 11]. The reason for this 'unboundedly' bad behavior (as a function of the number of processors) is that an extremely slow processor may bottleneck the entire system by spending a large amount of time on a task. This motivates the following class of heuristics. A *list schedule on the fastest i processors* has a priority list as above. The difference in the execution strategy is that the slowest $m-i$ processors are never used, and tasks are scheduled as if the only processors available were the fastest i processors.

To analyze this class of schedules the following definitions are necessary. A *chain* C is a sequence of tasks $C=(T_1,\ldots,T_l)$ with $T_i \in \mathcal{T}$ such that for all j, $1 \leq j < l$, $T_j < T_{j+1}$. C *starts* with task T_1. The *length* of C is $\sum_{j=1}^{l} \mu(T_j)$. The *height of a task* $T \in \mathcal{T}$ is the length of the longest chain starting at T. The *height of* $(\mathcal{T}, <, \mu)$ is the length of the longest chain starting at any task $T \in \mathcal{T}$.

While the notion of the height of a task is a static notion which is a property of $(\mathcal{T}, <, \mu)$, we also associate a dynamic notion of the height of a task with any schedule for $(\mathcal{T}, <, \mu)$. Specifically, let $\mathcal{S}$ be a schedule for $(\mathcal{T}, <, \mu)$, and let t be a time less than the finishing time of $\mathcal{S}$. Then the *height of the task T at time t* is equal to the length of the longest chain starting at T, where the length of the chain considers only the unexecuted time requirements. Similarly, the *height of* $(\mathcal{T}, <, \mu)$ *at time t* is the length of the longest chain starting at any task not yet completed $T \in \mathcal{T}$. Note that if a portion of a task has been finished at time t, then it contributes to the height only that proportion of the time requirement which has not yet been completed.

It is convenient to analyze schedules based on whether or not the height is decreasing during a given interval of time. One may plot the height of $(\mathcal{T}, <, \mu)$ as a function of time for a given schedule $\mathcal{S}$ and make the following observation. The height is a nonincreasing function which starts at the original height of $(\mathcal{T}, <, \mu)$ for $t=0$, and ends at height 0 at the finishing time of $\mathcal{S}$. If during an interval of time, the height was a monotonically decreasing function of time then that interval is called a *height reducing interval*. If during an interval the height is constant, the interval is called a *constant height interval*. Any schedule may be completely partitioned into portions executed during height reducing intervals, and portions executed during constant height intervals.

3. List schedules on the fastest i processors

The first portion of this section entails an analysis of the worst case performance of list schedules on the fastest i processors. Given a set of speeds $b_1, \ldots, b_m$, and given i, a bound will be obtained in terms of the parameters b_i's and i. The second portion of this section analyzes this bound more carefully, and indicates that for each set of processor speeds, an easily determined value of i causes the performance ratio to be no worse than $1+2\sqrt{m}$ times worse than optimal. A more complicated analysis then shows that in fact some value of i causes a ratio of no worse than $\sqrt{m}+O(m^{1/4})$. The final portion of this section provides examples which indicate that the performance bound is the correct order of magnitude. In particular, for certain sets of speeds, our heuristic and a class of related heuristics are as bad as $\sqrt{m-1}$ times worse than optimal.

It is easy to get a speed independent bound for a simple scheduling algorithm, but this bound is not very good. It is not hard to see that if only the fastest processor is used, and it is always used, that the resulting schedule is no worse than m times worse than optimal. This possibility is actually alluded to in [10, 11]. The bound of this section (which discusses a natural generalization of using only the fastest processor) is substantially better than this.

3.1. Performance bound on list schedules on the fastest i processors

The analysis approach that we use is to obtain two lower bounds, LB_1 and LB_2, on the finishing time of the optimal schedule for a given task system. Then an upper bound, UB, is obtained on the finishing time on the schedule of interest. The ratio $(UB/\max(LB_1, LB_2))$ is then an upper bound on the performance ratio of the schedule relative to optimal.

Define $B_i = \sum_{j=1}^{i} b_j$. Thus B_i is the *total processing power of the fastest i processors*. B_m is the total processing power of all the processors, and $B_1 = b_1$.

For a given task system, let μ denote the sum of the time requirements of the tasks in the system (by abuse of notation) and let h denote the (original) height of the system.

Lemma 3.1. *Let $(\mathcal{T}, <, \mu)$ be a task system to be scheduled on processors of different speeds. Let w_{opt} be the finishing time of an optimal schedule. Then $w_{\text{opt}} \geq \max(\mu/B_m, h/b_1)$.*

Proof. The most that any schedule can process in unit time is B_m units of the time requirement of the system. It thus follows immediately that $w_{\text{opt}} \geq \mu/B_m$.

A chain of length h requires at least time h/b_1 to be processed, even if the fastest processor is always used on the chain. It follows that $w_{\text{opt}} \geq h/b_1$. Combining these two bounds we obtain $w_{\text{opt}} \geq \max(\mu/B_m, h/b_1)$.

Lemma 3.2. *Let $(\mathcal{T}, <, \mu)$ be as in Lemma 3.1. Let w_i be the finishing time of a list schedule on the i fastest processors. Then $w_i \leq (\mu/B_i) + (h/b_i)$.*

Proof. To analyze the effectiveness of any list schedule on the fastest i processors, it is convenient to break up the schedule of interest into height reducing intervals and constant height intervals. The sum of the total duration of these intervals equals w_i.

Consider any constant height interval. Throughout the interval all of the fastest i processors are in use. We will prove this by contradiction. Let time t be a time within a constant height interval when fewer than i processors are in use. Consider the set of unfinished tasks that are at maximum height at time t. By definition, each of these tasks is executable (i.e. has no unfinished predecessors). Since not all of the fastest i processors are in use, and the schedule is a list schedule on the fastest i processors, it must be that all of these maximum height tasks are being executed. But then, it follows that the height of the task system in this interval is being reduced, contradicting the fact that this is a constant height interval.

By the above remarks, it follows that during each constant height interval, the processors of the machine are processing at least B_i units of the time requirement of the task system per unit time. Thus the total time spent on constant height intervals is at most μ/B_i.

Next, examine the height reducing intervals. At each point in time, some of the tasks being executed are at the maximum height. These as well as all other tasks being executed are being processed at the rate of at least b_i. Since the total height may be reduced by at most h throughout the schedule, it follows that the total amount of time spent on height reducing intervals is at most h/b_i. Together with the above bound on the amount of time in a constant height interval, one may conclude that $w_i \leq (\mu/B_i) + (h/b_i)$.

Actually it is easily shown that $w_i \leq ((\mu - h)/B_i) + (h/b_i)$ but this does not substantially improve the performance bound. This improvement follows from the fact that at least h units of the time requirement are executed during height reducing intervals leaving only $\mu - h$ for constant height intervals. It will be important to remember this improved bound for some numerical results in Section 3.2.

It follows from Lemmas 3.1 and 3.2 that:

$$\frac{w_i}{w_{\text{opt}}} \leq \frac{(\mu/B_i) + (h/b_i)}{\max(\mu/B_m, h/b_1)}. \tag{1}$$

Eq. (1) presents us with an opportunity to formally state the schedule that will be used. Given a task system $(\mathcal{T}, <, \mu)$, determine the total time requirement of all tasks (μ), and the height of the system (h). Compute the right-hand side of (1) for each value of $i = 1, \ldots, m$. The value of i that minimizes the expression is the number of processors that will be used. Devise any list schedule on the fastest i processors. In Section 3.2 it will be shown that the performance ratio of this schedule (relative to optimal) is at most $\sqrt{m} + O(m^{1/4})$.

The algorithm as stated above involves doing a separate calculation for each task system in order to determine how many processors to use. However, to get the $\sqrt{m}+O(m^{1/4})$ behavior, it is in fact possible to use the same number of processors independent of the task system (based only on the b_j's). By comparing the first lower bound on w_{opt} to μ/B_i and the second lower bound to h/b_i, (1) implies:

$$\frac{w_i}{w_{\text{opt}}} \leqslant \frac{(\mu/B_i)}{(\mu/B_m)} + \frac{(h/b_i)}{(h/b_1)} = \frac{B_m}{B_i} + \frac{b_1}{b_i}. \tag{2}$$

The second scheduling algorithm uses a list schedule on the fastest i processors where i minimizes the right-hand side of (2). In Section 3.2 it is shown that any resulting schedule is not worse than $\sqrt{m}+O(m^{1/4})$ times worse than optimal irrespective of the value of the b_j's.

Note that if the bound used on w_i is $w_i \leqslant ((\mu-h)/B_i)+(h/b_i)$, then one may obtain a bound on the algorithm of:

$$\frac{w_i}{w_{\text{opt}}} \leqslant \frac{B_m}{B_i} + \frac{b_1}{b_i} - \frac{b_1}{B_i}. \tag{3}$$

Thus, a third scheduling algorithm chooses i on the basis of (3). This improved algorithm does not have better asymptotic behavior, but does provide a better algorithm for small values of m. This will be explained in greater detail when numerical results are discussed. For asymptotic bounds we will use the algorithm generated by the simpler (2).

(*Notation*: The right-hand side of (2) will be denoted $E_i(b)$. The right-hand side of (3) will be denoted $E'_i(b)$.)

The bound of (3) with $i = m$ is the performance bound of Liu and Liu [10, 11]. For this reason, the choice of the value of i as the one which *minimizes* $E'_i(b)$ provides a bound which is at least as good as their bound. The derivation of (3) presented here is similar to the derivation of their bound.

In the remainder of this section the running times of the above algorithms is analyzed. Choosing the number of processors to use on the basis of (2) or (3) requires time $O(m)$ as follows. One needs $O(m)$ time to compute $B_1, B_2, \ldots, B_m$, $O(m)$ to compute $E_i(b)$ for each value of i, and $O(m)$ to minimize $E_i(b)$. Of course the value i need not be chosen separately for each task system to be scheduled on the same set of processors.

The actual scheduling of tasks once the processors are chosen requires time $O(n^2)$ if the proper data structures are used as follows. For each task T, a number pred(T) is maintained which represents the number of tasks that precede T that are not yet completed. The values pred(T) for $T \in \mathcal{T}$ are initialized in $O(n^2)$ time using the adjacency matrix for $<$. When a task T is completed, then if $T < T'$ the value pred(T') is decremented. It requires time $O(n)$ to update the pred function each time that a task is completed. If pred(T') is set to 0 when it is decremented, then T' is added to a list of executable tasks. The processor that had been executing the completed task, T, is then added to a list of idle processors (this list initially consists of the fastest i processors). The actual assigning of the tasks to the currently unassigned processors given these two lists can be done in constant time per task. Thus the leading term in the total time complexity is the $O(n^2)$ time required to do the initialization and to do n updates of the pred function (after the execution of each task). It is easy to verify that the schedule generated is a list schedule on the fastest i processors, as the fastest i processors are never unnecessarily idle.

To complete the analysis of the running time, we describe how to determine which of at most i currently executing tasks is finished next. This may be done in $O(\log m)$ time if a list of 'currently executing' tasks is maintained in order of their finish time. When a task is scheduled it is inserted into this list based on its finishing time. This must be done at most n times throughout the schedule. For $m < n$, $O(n \log m)$ is a low order term. For $n < m$, the insertion requires only $O(\log n)$ time, and $O(n \log n)$ is a low order term. Thus the total time complexity is $O(n^2 + m)$.

3.2. Calculation of speed-independent bound

This section analyzes Eq. (2) in three different ways. The first way provides an intuitive indication of which processors to use as a function of their relative speeds. Specifically, if the processors used are those whose rates are within a factor of $\sqrt{m}$ of the fastest processor, then list schedules on these processors are at most $1+2\sqrt{m}$ times worse than optimal. The second approach proves that one may always choose i such that the bound is $\sqrt{m} + O(m^{1/4})$. This complicated proof does not give any intuitive idea how to choose i in general, other than calculating $E_i(b)$ for each value of i and then minimizing. The third method of analysis is a calculation of the actual numerical bounds on the algorithm for small values of m when the correct choice of i is made. These bounds are better than $1 + 2\sqrt{m}$ and are more exact than a bound with an $O(m^{1/4})$ term.

Theorem 3.1. *Consider a set of m processors of different speeds. Then some value of $i (1 \leq i \leq m)$ has the property that for any task system (with optimal finishing time w_{opt}), and any list schedule on the fastest i processors for that task system (with finishing time w_i) $w_i/w_{\text{opt}} \leq 1 + 2\sqrt{m}$.*

Proof. Recall that $w_i/w_{\text{opt}} \leq (B_m/B_i) + (b_1/b_i)$. Choose i such that $\sqrt{m}b_i \geq b_1$ and $\sqrt{m}b_{i+1} < b_1$. Certainly some j satisfies $\sqrt{m}b_j \geq b_1$ (since b_1 satisfies it). If no j satisfies $\sqrt{m}b_{j+1} < b_1$, then choose $i = m$. From (2) and the choice of i:

$$\frac{w_i}{w_{\text{opt}}} \leq 1 + \frac{b_{i+1} + \cdots + b_m}{B_i} + \frac{\sqrt{m}b_i}{b_i}. \tag{4}$$

This follows from breaking up B_m into $B_i + b_{i+1} + \cdots + b_m$ and using the upper bound on b_1.

Now clearly $B_i \geqslant b_1$. Also using $\sqrt{m}b_j < b_1$ for $j \geqslant i+1$ (4) may be modified to obtain:

$$\frac{w_i}{w_{\text{opt}}} \leqslant 1 + \frac{(m-i)(b_1/\sqrt{m})}{b_1} + \sqrt{m}. \tag{5}$$

Since $(m-i) \leqslant m$ one may conclude that $w_i/w_{\text{opt}} \leqslant 1 + 2\sqrt{m}$.

Thus if the processors that are used are the ones whose speeds are within $\sqrt{m}$ of the fastest processor, we get a bound of $1+2\sqrt{m}$. In fact, this bound may be improved somewhat for the choice of i considered in the proof. We skip the details and proceed to give a more complicated proof which improves the bound of Theorem 3.1 by choosing i in a more intelligent manner.

Theorem 3.2. *Consider a set of m processors of different speeds. Then some value of $i(1 \leqslant i \leqslant m)$ has the property that for any task system (with optimal finishing time w_{opt}), and any list schedule on the fastest i processors for that task system (with finishing time w_i) $w_i/w_{\text{opt}} \leqslant \sqrt{m} + \mathrm{O}(m^{1/4})$.*

Proof. The value of i that is chosen is the one that minimizes $E_i(b)$. Let $f(m)$ be the supremum of $\min_i E_i(b)$, where the supremum is taken over $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_m > 0$. It will be shown that $f(m)$ is actually achieved by a particular set of speeds $b_1, \ldots, b_m$. Also, these speeds have the property that for every i, $f(m) = E_i(b)$. Using this fact one may conclude that $f(m) \leqslant \sqrt{m} + \mathrm{O}(m^{1/4})$.

Define $B \subset \mathbf{R}^m$ by $B = \{(b_1, \ldots, b_m) \in \mathbf{R}^m\colon\ b_1 \geqslant b_2 \geqslant \cdots \geqslant b_m \geqslant 0$ and $B_m = 1\}$. Note that B consists of every legal set of processor speeds (normalized to sum to 1), and some illegal sets (e.g. $b_m = 0$). For $b \in B$, with $b = (b_1, \ldots, b_m)$, define $g(b) = \min_i E_i(b)$. (Note that $g(b) \neq \infty$ since $E_1(b) = 1 + (1/b_1) < \infty$.) If $b_m \neq 0$, then $g(b)$ is a bound on the heuristic with processor speeds $b_1, \ldots, b_m$. Since B is compact and g is continuous, g attains a maximum at some particular point $b^* = (b_1^*, \ldots, b_m^*) \in B$. Note that $g(b^*) \geqslant f(m)$. Also, to show $g(b^*) = f(m)$ it suffices to show that $b_m^* \neq 0$.

It must be that $g(b^*) = E_m(b^*)$. To prove this, assume $g(b^*) \neq E_m(b^*)$ and define b' by $b_i' = b_i^*/(1+\varepsilon)$ and $b_m' = (b_m^* + \varepsilon)/(1+\varepsilon)$. For some ε $(0 < \varepsilon \ll 1)$, $b' \in B$ (if $b_m^* \neq b_{m-1}^*$), $E_i(b') > E_i(b^*)$ for $i < m$, and $E_m(b') < E_m(b^*)$. Since $E_m(b^*)$ was not the smallest E_i value for b^*, for sufficiently small ε, $g(b^*) < E_m(b')$. Since $E_i(b') > g(b^*)$ for every i, $g(b') > g(b^*)$, contradicting the fact that b^* maximizes g. (If $b_m^* = b_{m-1}^* = \cdots = b_{i+1}^* < b_i^*$, then b' as defined is not in B. In that case, one defines $b_j' = (b_j^* + (\varepsilon/(m-i)))/(1+\varepsilon)$ for $j = i+1, \ldots, m$ in order to preserve the decreasing nature of the vector b'. A similar proof then follows.)

To prove that for every i $E_i(b^*)$ is the same, it suffices to consider the case that $g(b^*) = E_m(b^*) = \cdots = E_{k+1}(b^*) < E_k(b^*)$. We will define a sequence of vectors b', b'', b''' so that each successive vector has the various E_i values at least as large as $g(b^*)$ but agreeing with g at one less value of i. That is $E_{k+1}(b') > g(b^*)$, $E_{k+2}(b'') > g(b^*)$, etc. Finally, we will obtain $E_m(b) > g(b^*)$ for some vector b with either $g(b) > g(b^*)$

or $g(b)=g(b^*)$. In the first case it contradicts the fact that b^* maximizes g. In the second case it contradicts $g(b^*)=E_m(b^*)$ since b then maximizes g and $g(b)\neq E_m(b)$.

The vector b' is defined by $b'_i=b^*_i$ for $i\neq k$, $k+1$, $b'_k=b^*_k+\varepsilon$ and $b'_{k+1}=b^*_{k+1}-\varepsilon$. It is easy to verify that both $E_{k+1}(b')$ and $E_k(b')$ exceed $g(b^*)$ for small ε. Also $E_i(b')=E_i(b^*)$ for $i\neq k$, $k+1$. We now show $b'\in B$. From $E_{k+2}(b^*)=E_{k+1}(b^*)$ we have $b^*_{k+1}>b^*_{k+2}$. Thus $b'_{k+1}>b'_{k+2}$ for small ε. However, $b^*_k=b^*_{k-1}$ is possible which would imply $b'\notin B$. In that case, the ε is not added to b^*_k, but rather a total of ε is added to the processors whose rates equal b^*_k. The vectors b'', b''', ... are defined in a similar manner. The final vector has $E_m(b)>g(b^*)$. Since $g(b^*)=E_m(b^*)$, the vector b cannot maximize g. But $E_i(b)\geq g(b^*)$ for every $i=1,\dots,m$. This contradicts the fact that b^* maximizes g.

It follows from the fact that $E_i(b^*)=g(b^*)$ for every i that $b^*_m\neq 0$ and thus $g(b^*)=f(m)$.

The bound of $\sqrt{m}+O(m^{1/4})$ will be proved by using the fact that $f=f(m)=E_i(b^*)$ for each value of i. Using $f=E_i(b^*)$ and $f=E_1(b^*)$ one may conclude that $b^*_i=B^*_i/(f-1)(fB^*_i-1)$ since $b^*_i=b^*_1B^*_i/(fB^*_i-1)$ and $b^*_1=1/(f-1)$. This in turn proves that $b^*_i=(1/f(f-1))(1+1/(fB^*_i-1))$. Thus for $j>i$ $b^*_j\leq(1+1/(fB^*_i-1))/f(f-1)$. From this it follows that:

$$b^*_{i+1}+\cdots+b^*_m=1-B^*_i\leq(1+\varepsilon)(m-i)/f(f-1),\quad \text{where } \varepsilon=1/(fB^*_i-1). \tag{6}$$

In order to use the above equation to get a bound on f, a few other facts are needed. Note that $f>\sqrt{m}$. This can be shown by considering the vector b' which is a normalized version of the vector $(\sqrt{m-1},1,1,\dots,1)$ since $g(b')>\sqrt{m}$. Note also that $b^*_1\leq\sqrt{1/m}$. This follows from the fact that $E_1(b^*)=E_m(b^*)$ which implies that $b^{*2}_1=b^*_m$. Since $mb^*_m\leq 1$ the upper bound on b^*_1 follows. Thus for $j=1,\dots,m$ $b^*_j\leq b^*_1<1/\sqrt{m}$ and the successive sums $B^*_1, B^*_2,\dots,B^*_m$ are spaced apart by a distance of at most $1/\sqrt{m}$. Thus, for some value of i, $B^*_i=rm^{-1/4}$ for some r between $\sqrt{2}$ and $1+\sqrt{2}$ (for sufficiently large m).

Let $B^*_i=rm^{-1/4}$. Then $1+B^*_i\geq 1+\varepsilon$ using the expression for ε in (6). This follows from the fact that

$$B^*_i/\varepsilon=fB^{*2}_i-B^*_i\geq\sqrt{m}r^2m^{-1/2}-B^*_i=r^2-B^*_i$$

using $f>\sqrt{m}$ and $B^*_i=rm^{-1/4}$. Since $r^2\geq 2$ and $B^*_i\leq 1$, $B^*_i/\varepsilon\geq r^2-B^*_i\geq 2-1=1$ and thus $B^*_i\geq\varepsilon$. Using (6) and $1+B^*_i\geq 1+\varepsilon$ we have

$$f(f-1)\leq(m-i)(1+B^*_i)/(1-B^*_i)\leq(m-i)(1+rm^{-1/4})(1+2rm^{-1/4})$$

for sufficiently large values of m (the last inequality is obtained by expanding $1/(1-B^*_i)$). But then (by further increasing the right-hand side) $(f-1)^2\leq m+4rm^{3/4}+4r^2\sqrt{m}$ which yields $f\leq\sqrt{m}+2rm^{1/4}+1$.

While Theorem 3.2 provides a better asymptotic estimate of the performance of the algorithm than Theorem 3.1, it does not give a better bound for practical situations. In principle the $O(m^{1/4})$ term may be the dominating factor for the small values of m that arise in practice. For that reason, it is important to get a more meaningful bound for small values of m. A third way of evaluating the heuristic based on (2) is thus presented, which gives numerical bounds on the algorithm for small values of m. This also will give an intuitive idea as to the growth rate of $f(m)$.

Recall that the heuristic takes its worst value at the vector b^* with the property that $E_i(b^*)$ is the same for every i. From $E_1(b^*) = E_m(b^*)$ with $B_m^* = 1$, we get $b_m^* = b_1^{*2}$. From $E_i(b^*) = E_1(b^*)$ we get for $1 < i < m$,

$$b_i^* = \frac{b_1^{*2} B_i^*}{B_i^*(b_1^* + 1) - b_1^*}. \tag{7}$$

Using $B_i^* = 1 - (b_{i+1}^* + \cdots + b_m^*)$, we note that the above equation is in terms of b_1^* and $b_{i+1}^*, \ldots, b_m^*$. Hence each b_i^* can be determined inductively as a function of b_1^*. Using $1 = b_1^* + \cdots + b_m^*$, we can solve for b_1^*. Solving this equation and computing $1 + 1/b_1^*$ gives a bound on the algorithm. It is not hard to show that there is a unique solution to this equation subject to $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_m > 0$.

This calculation was done on the MACSYMA system which generated the expressions to be solved and then solved them. The indication for this small sample of data is that $\sqrt{m} + O(\log m)$ might in fact be an accurate bound. The value $f(m)$ for the range of values considered seems to be bounded by $\sqrt{m} + 0.21 \log_2 m + 1$. In fact, not only is $f(m)$ bounded by this expression (in the range we considered), but it seems to grow slower than this expression. The results together with other key quantities are given in Table 1.

Table 1

m	$f(m)$	$\sqrt{m}$	$\sqrt{m} + m^{1/4}$	$1 + 2\sqrt{m}$	$\sqrt{m} + 0.21 \log_2 m + 1$
2	2.62	1.41	2.60	3.82	2.62
3	3.06	1.73	3.05	4.46	3.06
4	3.41	2	3.41	5	3.42
5	3.71	2.24	3.74	5.48	3.73
6	3.98	2.45	4.02	5.90	3.99
7	4.22	2.65	4.28	6.30	4.24
8	4.44	2.83	4.51	6.66	4.46
9	4.64	3	4.73	7	4.67
10	4.83	3.16	4.94	7.32	4.86
50	9.14	7.07	9.73	15.14	9.26
100	12.24	10	13.16	21	12.40
500	24.98	22.36	27.09	45.72	25.24
1000	34.41	31.62	37.25	64.25	34.71
5000	73.88	70.71	79.12	142.42	74.29
10 000	103.33	100	110	201	103.79

Note that $f(m)$ does seem to be growing faster than $\sqrt{m} + O(1)$ although this cannot be proven by such numerical studies.

The above results were obtained using the bound of (2). An important purpose of these results is to show how $f(m)$ behaves. An additional reason for this calculational exercise is to get as good a bound as possible on the heuristic for small values of m. This goal is furthered if we use the slightly more complicated bound given by (3). In our analytic studies the b_1/B_i term was ignored since it does not improve the asymptotic results (in particular it is always less than 1). Nevertheless, for small values of m it is a significant portion of the bound. The second table gives a bound on the algorithm without neglecting this extra term. (The same technique was used for generating Table 2 as was used for generating Table 1. To use this technique, it must first be shown that a bound on the algorithm is obtained by analyzing the vector $b^* \in B$ which has the property that $E_i'(b^*)$ is the same for each value of i. This is a simple exercise using the technique of the proof of Theorem 3.2.)

Using the bound of (3) gives a result which is about 0.7 or 0.8 better than the bound of (2)—a substantial saving for small m. For large values of m the improvement is less significant due to the large value of either bound.

Table 2

m	bound on the algorithm
2	1.75
3	2.25
4	2.65
5	2.97
6	3.25
7	3.50
8	3.73
9	3.94
10	4.14

Intuitively it seems quite wasteful *never* to use certain processors. It is an open question to determine how to use the slow processors in order to provide a quantitatively better performance ratio. There are certain simple safe techniques that one may use which do not harm the performance ratio. For example, one may first determine a list schedule on the fastest i processors. Then, if a slow processor is free at a particular time, and an executable task is not being executed, and furthermore the finishing time of the task will be later with the current schedule than the time that our slow processor could finish it, then it is safe to assign the task to the slow processor.

The fact that this procedure is not harmful may be easily seen. Since the finishing time of the chosen task is earlier in the new schedule than in the original schedule, no task needs to be finished later than in the original schedule. While it is easy to determine such safe uses for the slow processors, we have been unable to determine any methods that guarantee faster behavior.

3.3. *Achievability of the performance bound*

In this section it is shown that the results of Section 3.2 are asymptotically correct. This is done by demonstrating that for a certain set of processor speeds and a specific task system, the performance ratio of a list schedule on the fastest i processors (for *any* $i = 1, \ldots, m$) may be as large as $\sqrt{m-1}$. The fact that this example shows that any choice of i has the potential of being $\sqrt{m-1}$ times worse than optimal is significant. It tells us that no sophisticated way of choosing i provides better than $\sqrt{m}$ behavior if once i is chosen a list schedule is the only added feature of the heuristic.

Consider the situation, where $b_1 = \sqrt{m-1}$ and $b_i = 1$ for $i > 1$. Consider the task system of $2n$ tasks as diagrammed in Fig. 1. A node represents a task and an arrow represents a precedence dependence. The time requirement of each of the n tasks in the long chain is $\sqrt{m-1}$. The time requirement of the other n tasks is $m-1$. An asymptotically optimal schedule proceeds as follows. P_1 executes every task in the long chain. Each task in the long chain requires unit time on P_1. Meanwhile, $P_2, \ldots, P_m$ execute the tasks that are not in the long chain. Each of these processors requires time $m-1$ for one of the tasks. If $n = m-1$, then the long chain requires time $m-1$, but P_m will not finish its task until $2m-3$ units of time have passed since its task is not executable until $m-2$ units of time elapse. For any value of n the finishing time is similarly bounded by $n + m - 2$.

To discuss the fact that this task system may be executed inefficiently, no matter how many processors are used, consider two situations. The first is the case that one attempts to schedule the system on the fastest processor. The second is the situation that the processing is done on i processors for any $i > 1$.

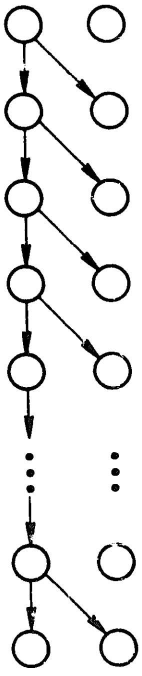

Fig. 1.

If only the first processor is used, then there is not enough processing power to execute this task system efficiently. Specifically, the total amount of time requirement of this task system is $n(m-1+\sqrt{m-1})$. With only $\sqrt{m-1}$ processing power available, the finishing time must be at least $n(1+\sqrt{m-1})$. For large values of n, this provides a performance ratio of approximately $1+\sqrt{m-1}$ times worse than optimal.

Consider the scheduling of this system on i processors for $i>1$. A 'bad' list schedule uses P_1 on the 'non-long chain' tasks, and P_2 on the chain tasks. After time $\sqrt{m-1}$, P_1 finishes the first non-chain element, and P_2 finishes the first chain element. Repeating this stategy for each pair of tasks requires time $\sqrt{m-1}$ for each pair. Thus the total time for the bad schedule is about $n\sqrt{m-1}$ and the ratio between the finishing times of the 'bad' schedule and the optimal schedule approaches $\sqrt{m-1}$ for large n. Note that no matter how many processors one attempts to use, a bad list schedule only allows two processors to be used.

The fact that $m-1$ of the m processors have the same speed in this example is important. In [3] it is shown that with independent tasks and almost identical processors (in the sense of $m-1$ of them being identical) improved algorithms may be obtained.

Recall that the upper bound on the algorithm was $\sqrt{m}+O(m^{1/4})$. It has not been shown that this is the best upper bound on the algorithm. However, whatever the numerical value of $f(m)=\max_b(g(b))$ is, a bound on the heuristic of Section 3.1 may be obtained in terms of $f(m)$. Consider a set of m unordered tasks, the ith having time requirement b_i^*. The optimal schedule requires unit time. Using only the fastest processor (a valid choice with the algorithm as presented) requires time $1/b_1^*$. But $f(m)-1=1/b_1^*$. Thus $f(m)$ is almost achievable (whatever $f(m)$ is). This means that an exact bound on the algorithm may be obtained by solving the mathematical problem of determining $f(m)$, without any need to look at more task systems.

Consider the related heuristic of trying to minimize

$$E_i'(b^*)=\frac{1}{B_i^*}+\frac{b_1^*}{b_i^*}-\frac{b_1^*}{B_i^*}.$$

In that case $E_1'(b^*)=1/b_1^*$. Recall that at the vector b^* that maximizes $\min_i E_i'(b^*)$, $E_i'(b^*)$ is the same for every i, and using only the fastest processor is a valid choice. In that case, $1/b_1^*$ for this vector b^* is an exact upper and lower bound.

To give a concrete example, consider $m=2$. $E_1'(b)=1/b_1$ and $E_2'(b)=1+(b_1/b_2)-b_1$. Solving $E_1'(b)=E_2'(b)$ provides $b_1=0.57$ and $b_2=0.43$. Let $\mu(T_1)=0.57$, $\mu(T_2)=0.43$ and $<$ = empty. Then the optimal schedule requires unit time. The algorithm presented here may choose to use only the fastest processor and require time $1/0.57=1.75$. This agrees with the bound predicted by solving the maximization problem.

4. Preemptive scheduling

This section discusses preemptive scheduling. Preemptive scheduling permits the temporary suspension of the execution of a task. When the task is continued, only the unexecuted portion needs to be finished, and there is no penalty for the temporary suspension. Formally:

A *preemptive schedule* for $(\mathscr{T}, <, \mu)$ on a set of processors $\mathscr{P}$ is a total function $\mathscr{S}$ that maps each task $T \in \mathscr{T}$ to a finite set of interval, processor pairs. If $\mathscr{S}(T) = \{([i_1, j_1], Q_1), ([i_2, j_2], Q_2), \ldots, ([i_l, j_l], Q_l)\}$, then

(1) $i_p, j_p \in \mathbf{R}^+$ for $p = 1, \ldots, l$,

(2) $i_p \leqslant j_p$ for $p = 1, \ldots, l$ and $j_p \leqslant i_{p+1}$ for $p = 1, \ldots, l-1$,

(3) $Q_p \in \mathscr{P}$ for $p = 1, \ldots, l$.

For $i_p \leqslant t < j_p$ *T is being executed on processor Q_p at time t.* The time i_1 is the *starting time* of T, and the time j_l is the *finishing time* of T.

A *valid* preemptive schedule for $(\mathscr{T}, <, \mu)$ *on a set of processors* $\mathscr{P}$ is a preemptive schedule for $(\mathscr{T}, <, \mu)$ with the properties:

(1) for all $t \in \mathbf{R}^+$, if two tasks are both being executed at time t, then they are being executed on different processors at time t;

(2) for $T, T' \in \mathscr{T}$, if $T < T'$, the starting time of T' is not smaller than the finishing time of T;

(3) for $T \in \mathscr{T}$ (with $\mathscr{S}(T)$ as above), $\mu(T) = ((j_1 - i_1)/r(Q_1)) + \cdots + ((j_l - i_l)/r(Q_l))$ where $r(Q_i)$ is the rate of Q_i (i.e. if $Q_i = P_j$, then $r(Q_i) = b_j$).

Condition three asserts that each task is processed exactly long enough to complete its time requirement.

The performance of the *maximal usage* heuristic is discussed. A *maximal usage preemptive schedule* is a valid preemptive schedule satisfying the following two requirements:

(1) whenever i tasks are executable, then $\min(m, i)$ tasks are being executed (a task is executable if all its predecessors have been finished, but the task itself has not been finished);

(2) whenever i processors are being used, the fastest i processors are in use.

It is easy to see how to transform any schedule $\mathscr{S}$ into a maximal usage schedule that has a finishing time at least as small as that of $\mathscr{S}$.

Maximal usage preemptive schedules were studied in [7]. It was shown that any maximal usage schedule has a finishing time not worse than $\frac{1}{2} + \sqrt{m}$ times worse than optimal.

In this section a performance bound on maximal usage schedules is obtained by appealing directly to the results of Section 3. Fix a set of processors. Consider (2) in Section 3.1. It suffices to show that (2) (when interpreted as a bound on the performance of any maximal usage schedule) applies to any task system scheduled with any maximal usage schedule, and any value of i. From this, one may conclude that for any task system and any maximal usage schedule (with finishing time w),

$w/w_{\text{opt}} \leq \sqrt{m} + O(m^{1/4})$. This proof follows by applying the bound of (2) and using Theorem 3.2. For some value of i the right-hand side of (2) is bounded by $\sqrt{m} + O(m^{1/4})$. Note that in this context w_{opt} represents the finishing time of the optimal *preemptive* schedule.

It suffices to show that w_{opt} satisfies the lower bound of Lemma 3.1 and w satisfies the upper bound of Lemma 3.2. The former is immediate, since the lower bound did not consider the fact that nonpreemptive schedules were used.

To get the upper bound on w given in Lemma 3.2, break up all intervals of any maximal usage schedule into two types of intervals. One type of interval is when at least i processors are being used, and the second type is when fewer than i processors are in use. Clearly, one may use at least i processors for at most a total of μ/B_i units of time. Also, the intervals during which fewer than i processors are in use are height reducing intervals. These height reducing intervals decrease the height by a rate of at least b_i per unit time. Lemma 3.2 follows and thus one may conclude:

Theorem 4.1. *Let $(\mathcal{T}, <, \mu)$ as above. Let w be the finishing time of any preemptive maximal usage schedule, and let w_{opt} be the finishing time of an optimal preemptive schedule. Then $w/w_{\text{opt}} \leq \sqrt{m} + O(m^{1/4})$.*

5. Extensions and conclusion

In [8, 12] the scheduling of jobs on processors of different types is considered. In this model, tasks and processors are both partitioned into sets of different *types* and a task may be executed only by a processor of the same type. In [8], an analysis of list schedules was presented for the situation that within each type, processors were of different speeds. In [9] the methods of this paper are extended to get speed independent bounds on certain scheduling algorithms for this machine environment. The techniques are more complicated and the bound of Theorem 3.1 is generalized in two different ways. Each generalization may produce a better bound, based on the number of processors of each type. The maximal usage preemptive schedules for typed task systems are also considered.

The algorithms presented in this paper are examples of scheduling algorithms that violate the naive 'greedy' heuristic of trying to schedule as many tasks as possible at each point in time. In this context methods have been developed for deciding when to be greedy and how greedy to be. It would be interesting to obtain similar algorithms for other situations, for example where the processors are identical. Also, one would hope to find improvements in the algorithms presented, possibly by making use of the time requirements of the tasks or the nature of the partial order. Since any algorithm that does not use the time requirement of a job before scheduling it is at least $\sqrt{m}$ times worse than optimal [2], the improvements are likely to be in areas which use the time requirements of the tasks as a factor in the scheduling algorithm.

Acknowledgment

The author wishes to express his gratitude to E. Davis and M. Rabin for helpful discussions and in particular for their suggestions on how to approach the proof of Theorem 3.2. Also, to A. Meyer for several helpful comments. Finally, to D. Kessler for programming the MACSYMA calculations.

References

[1] Y. Cho and S. Sahni, Bounds for list schedules on uniform processors, University of Minnesota TR78-13 (1978).

[2] E. Davis and J.M. Jaffe, Algorithms for scheduling tasks on unrelated processors, *J. ACM*, to appear.

[3] T. Gonzalez, O.H. Ibarra and S. Sahni, Bounds for LPT schedules on uniform processors, *SIAM J. Comput.* **6**(1) (1977) 155–166.

[4] R.L. Graham, Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.* **17** (1969) 263–269.

[5] E. Horowitz and S. Sahni, Exact and approximate algorithms for scheduling nonidentical processors, *J. ACM* **23**(2) (1976) 317–327.

[6] E.C. Horvath, S. Lam and R. Sethi, A level algorithm for preemptive scheduling, *J. ACM* **24**(1) (1977) 32–43.

[7] J.M. Jaffe, An analysis of preemptive multiprocessor job scheduling, *Math. Operations Res.*, to appear.

[8] J.M. Jaffe, Bounds on the scheduling of typed task systems, *SIAM J. Comput.*, to appear.

[9] J.M. Jaffe, Efficient scheduling of tasks without full use of processor resources, MIT, Laboratory for Computer Science, Technical Memo 122 (1979).

[10] J.W.S. Liu and C.L. Liu, Bounds on scheduling algorithms for heterogeneous computing systems, TR No. UIUCDCS-R-74-632 Dept. of Comp. Sci., Univ. of Illinois (1974).

[11] J.W.S. Liu and C.L. Liu, Bounds on scheduling algorithms for heterogeneous computing systems, in: J.L. Rosenfeld, Ed., *Information Processing 74* (North-Holland, Amsterdam, 1974) 349–353.

[12] J.W.S. Liu and C.L. Liu, Performance analysis of multiprocessor systems containing functionally dedicated processors, *Acta Informat.*, **10**(1) (1978) 95–104.

[13] R.R. Muntz and E.G. Coffman Jr., Optimal preemptive scheduling on two-processor systems, *IEEE Trans. Comput.* **18**(11) (1969) 1014–1020.

[14] R.R. Muntz and E.G. Coffman Jr., Preemptive scheduling of real time tasks on multiprocessor systems, *J. ACM* **17**(2) (1970) 324–338.