MIT/LCS/TM-137

ALGORITHMS FOR SCHEDULING TASKS ON UNRELATED PROCESSORS

Ernest Davis

Jeffrey M. Jaffe

June 1979

ALGORITHMS FOR SCHEDULING TASKS ON UNRELATED PROCESSORS

Ernest Davis and Jeffrey M. Jaffe

June 1979

This report was prepared with the support of a National Science Foundation graduate fellowship, and National Science Foundation grant no. MCS77-19754

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

Algorithms for scheduling tasks on unrelated processors

Ernest Davis [+] and Jeffrey M. Jaffe [*]

Abstract. Several algorithms are presented for the nonpreemptive assignment of n independent tasks to m unrelated processors. One algorithm requires polynomial time in n and m, and is at most $2\sqrt{m}$ times worse than optimal in the worst case. This is the best polynomial time algorithm known for scheduling such sets of tasks. An algorithm with slightly better worst case performance requires polynomial time in n but exponential time in m. This is the best algorithm known that requires time $O(n\log(n))$ for every fixed value of m.

Keywords. Nonpreemptive schedules, worst case finishing time, performance ratio, unrelated processors, largest processing time.

1. Introduction

This paper presents a number of polynomial time algorithms for the scheduling of a set of n independent tasks on m processors of different speeds. The processors are unrelated in the sense that there is no notion of a fast processor always requiring less time than a slow processor, irrespective of the task being executed. Rather, the time required for the execution of a task on

---

[+] CE Lummus, Inc., Bloomfield, NJ

[*] Laboratory for Computer Science, MIT, 545 Tech. Sq., Cambridge, MA 02139

This report was prepared with the support of a National Science Foundation graduate fellowship, and National Science Foundation grant no. MCS77-19754.

a processor is a function of both the task and the processor. This models the situation that general purpose processors have specialized capabilities that permit them to execute certain tasks more efficiently than others. An example of this might be in a distributed system where the time requirement of a task on a processor may depend on communication costs.

The decision problem of determining whether a given set of tasks can be assigned with finishing time smaller than a given bound is NP-complete [6]. As a result it seems unlikely that an algorithm can be found which runs in polynomial time and always produces the optimal assignment [3]. It is thus worthwhile to investigate approximation algorithms.

Polynomial time approximation algorithms for such sets of tasks were first studied by Ibarra and Kim in [6]. Five algorithms were presented, each of which was guaranteed to be at most m times worse than optimal in the worst case. In addition, four of the five were proved to be exactly m times worse than optimal in the worst case. The fifth algorithm was left as an open problem - its effectiveness was shown to be between 2 and m times worse than optimal. (In Section 8 an example is presented which indicates that this algorithm is at least $1+\log_2(m)$ times worse than optimal in the worst case. Thus the gap left in [6] is somewhat tightened, but is still left open.)

The first new algorithm that we present is at most $2.5\sqrt{m}$ times worse than optimal in the worst case. Thus it may not be as good as the fifth algorithm of [6], but it is provably better than the other four, and may in fact be better than all five. The running time of this algorithm is $O(mn\log(n))$. We also show that there are examples for which the algorithm is as bad as $2\sqrt{m}$ times worse than optimal, indicating that the analysis is tight up to a factor of 1.25. This algorithm is somewhat similar to that of [7] in that processors do not execute tasks for which they are very inefficient. This algorithm is best possible (up to a constant factor) among algorithms using a certain restricted class of heuristics.

The second new algorithm is a modification of the first, which adds a largest processing time (LPT) heuristic. This algorithm is at most $(1+\sqrt{2})\sqrt{m}$ times worse than optimal and also runs in polynomial time. The worst example known for this heuristic is $\sqrt{m}$ times worse than optimal, and we believe that this heuristic is actually more of an improvement over the original algorithm than the worst case bound suggests. The LPT heuristic has been a useful heuristic in situations where the processors are identical [4].

The third algorithm is a different modification of the first with a substantially longer running time. Whereas the first two require polynomial time in terms of both the number of tasks and the number of processors, this one requires exponential time in terms of the number of processors. When assigning tasks on a bounded number of processors, however, the algorithm runs in polynomial time. The worst case behavior is at most $1.5\sqrt{m}$ times worse than optimal. Horowitz and Sahni [5,12] devise algorithms of time complexity $O(n^{2m}/\epsilon)$ whose worst cases are within $1+\epsilon$ of optimal. Our algorithm requires time $O(m^m+mn\log(n))$. Thus its running time is less sensitive to large numbers of tasks executed on moderate numbers of processors.

All three of these algorithms may be varied in trivial ways to get performance bounds which are better than the above bounds by a small constant factor. While we are able to obtain slightly better performance bounds with these modifications, we do not believe that the resulting algorithms are actually more effective than the original algorithms.

One final result that we present is an additional algorithm which is m

times worse than optimal in the worst case. This algorithm has the useful property that when extended to an algorithm for scheduling tasks subject to precedence constraints, i.e., partially ordered tasks, it is still at most m times worse than optimal.

There has been other work on the scheduling of tasks on unrelated processors. Preemptive scheduling of tasks on unrelated processors is studied in [10,13]. In [1,2] a different optimality criterion is studied. Special cases in which certain processors cannot execute certain tasks are studied in [8,9,11].

2. The scheduling algorithm

Definitions.

A *task system of n tasks and m processors* is an $n \times m$ matrix μ with entries in $\mathbb{R}^+ \cup \{\infty\}$ for $n,m \geq 1$ such that for every t there is a p such that $\mu(t,p) \neq \infty$ $(1 \leq t \leq n)$.

The value $\mu(t,p)$ is the *time requirement* of the tth task on the pth processor $(1 \leq t \leq n,\ 1 \leq p \leq m)$.

The execution of jobs by processors is modelled by the notion of an assignment function for a task system. An *assignment function A* is a map $A{:}\{1,\ldots,n\} \rightarrow \{1,\ldots,m\}$ such that $\mu(t,A(t)) \neq \infty$ for $t=1,\ldots,n$. If $A(t)=p$ we will often say that *task t is assigned to processor p*. In general we will often refer to the index t as the tth task, and the index p as the pth processor. The *finishing time of an assignment, A,* (denoted $f(A)$) is defined by:

$$f(A) = \max_{1 \leq p \leq m} \sum_{t:A(t)=p} \mu(t,p)$$

An *optimal assignment* is any assignment that minimizes the finishing time. For two assignments A and B, the *performance ratio of A to B* is $f(A)/f(B)$.

Intuitively, the assignment function assigns jobs to processors. The finishing time intuitively represents the largest amount of time needed by any of the processors to execute the tasks assigned to it.

It is useful to associate a starting time function s, with a given assignment function A. Intuitively, for $1 \leq t \leq n$ the value $s(t)$ represents the time at which processor $A(t)$ begins to execute task t. Formally, a *starting time function* s, is a map $s:\{1,\ldots,n\} \to \mathbb{R}$ satisfying conditions (a) and (b) below. The value $s(t)$ is called the *starting time* of the task t. Task t is *being executed on processor p at time x* providing that $p=A(t)$ and $s(t) \leq x < s(t)+\mu(t,p)$.

A starting time function satisfies:

(a) For $p=1,\ldots,m$, at most one task is being executed at any time on processor p.

(b) For $p=1,\ldots,m$, if $0 \leq x < \Sigma_{t:A(t)=p}\, \mu(t,p)$ then at least one task is being executed at time x on processor p.

Intuitively, the first condition forces all tasks assigned to a processor to be executed sequentially, and the second condition prevents any idle periods on the processor.

By abuse of notation $f(t)=s(t)+\mu(t,A(t))$ is called the *finishing time* of the task t. Similarly, if $T \subset \{1,\ldots,n\}$, $f(T)=\max_{t \in T} f(t)$.

The matrix μ associates m possible time requirements with each task. The *best time* of the t^{th} task, (denoted $b(t)$), is the smallest of these m values (i.e. $b(t)=\min_p \{\mu(t,p)\}$). The *efficiency* of the p^{th} processor on the t^{th} task (denoted $ef(t,p)$) is $b(t)/\mu(t,p)$. Note that the maximum efficiency is one.

The following algorithm devises an assignment A and a starting time function s for a given task system.

Algorithm 1.

1. For $1 \leq p \leq m$ sort the n values $ef(t,p)$, $1 \leq t \leq n$. Set $sum_p=0$ for $p=1,\ldots,m$. Designate all processors as being "active" and all tasks as being "unassigned".

2. Find any value of p such that sum_p is minimal among active processors. (Note that there must always be such a p.)

3. Find the task, t with largest value of $ef(t,p)$ among unassigned tasks. If there are no such tasks then HALT. If $ef(t,p) \geq 1/\sqrt{m}$ go to Step 4. Otherwise designate p as being inactive and go to Step 2.

4. Define $A(t)=p$. Designate t as being assigned. Define $s(t)=sum_p$. Set $sum_p=sum_p+\mu(t,p)$. Go to Step 2.

It may be noted that A is an assignment function. The algorithm terminates when there are no unassigned tasks remaining. Note that at termination some processor is active. For, the processor that has the best time on the last task assigned could never have been deactivated.

Since each iteration of the main loop (steps 2-4) either assigns a task or deactivates a processor, the algorithm terminates after $n+m$ iterations.

It may be noted that this algorithm can be applied as a run time scheduler even if the absolute time requirements of each of the jobs are not known in advance, as long as the efficiencies are known. The only place that the time requirements were needed was in determining which processor is the next to be assigned a task. If this decision is made at run time (i.e., after a processor completes all tasks already assigned to it), then the time requirements are not needed while the assignment is being made.

The assignment A and starting time function s determined by Algorithm 1 satisfy the following three conditions:

(1) If $A(t)=p$ then $ef(t,p) \geq 1/\sqrt{m}$.

(2) If $s(t)>s(u)$, then $ef(t,A(u)) \leq ef(u,A(u))$ $(1 \leq t,u \leq n)$.

(3) If $\Sigma_{t:A(t)=p}\, \mu(t,p)<s(u)$, then $ef(u,p)<1/\sqrt{m}$.

Intuitively, condition one indicates that a task is executed on a processor only if the processor is "somewhat" efficient for the task. The second condition ensures that if a task u is assigned at an earlier time than t, it must be that processor $A(u)$ is more efficient on u than t. The third condition prevents a processor from stopping as long as it is still somewhat efficient for some unstarted task.

The fact that Algorithm 1 satisfies condition (1) is immediate from the way tasks are assigned. Condition (2) follows from the fact that if $s(t)>s(u)$, then the fact that u (and not t) is assigned to $A(u)$ implies that u must have at least as high an efficiency on $A(u)$ as t. Condition (3) also follows immediately from the way tasks are assigned.

In the sequel, the only facts used about Algorithm 1 are conditions (1), (2), and (3) above. Thus the analysis of Algorithm 1 applies to any algorithm that follows these principles. In Section 6 we combine Algorithm 1 with a different heuristic which preserves (1), (2), and (3), and thus the entire analysis applies to the modified algorithm.

To analyze the running time note that the presorting requires time $O(n\log(n))$ for each value of p (if sorting is done with comparisons). Consider the total time spent on iterations in which the value p is chosen in Step 2. Determining if a given task is unassigned requires time $O(\log(n))$ if that information is stored as a bit array. Thus, steps 3 and 4 of all iterations in which a particular p is chosen require time at most $O(n\log(n))$ (assuming that the list of tasks sorted by $ef(t,p)$ is maintained with a pointer to the task that will be looked at next). If a data structure is maintained which keeps sum_p sorted, the $m+n$ iterations of step 2 require at most time $O((m+n)(\log(m)))$. If $m>n$, a slightly different data structure permits the execution of step 2 in time $O(m\log(n))$. Thus the total running time is $O(mn\log(n))$.

3. Analysis of Algorithm 1

To analyze Algorithm 1 for a given task system, we fix a particular assignment A and starting function s consistent with (1), (2), and (3). Certain subsets of $\{1,\ldots,n\}$ will be defined based on s,A, and an optimal assignment function B. Let z be the last (or one of the last if there are ties) task, to be completed, i.e., $f(z)=f(A)$. Consider a value p such that $\mu(z,p)=b(z)$, i.e., processor p is one of the processors that is most efficient on z. Without loss of generality assume that $p=1$, i.e., that processor 1 is most efficient on z.

Definition. Let K consist of those tasks executed on the first processor with starting time earlier than $s(z)$, i.e. $K=\{t\colon A(t)=1 \text{ and } s(t)<s(z)\}$.

By condition (3) the first processor may not finish earlier than $s(z)$. Furthermore, all tasks of K must have efficiency one on the first processor or else condition (2) is violated.

The set K will be partitioned into two sets. The assignment of each task in B determines the set to which it belongs.

Definition. Let $L=\{t\colon t\in K \text{ and } ef(t,B(t))\geq 1/\sqrt{m}\}$.

Thus L consist of those tasks executed on processor 1 before time $s(z)$, that are assigned somewhat efficiently in B. The second subset of K will be denoted $K-L$.

Let $A^{-1}(\{2,\ldots,m\})$ be the set of tasks that A does not assign to the first processor, i.e. $A^{-1}(\{2,\ldots,m\})=\{t{:}A(t)\neq 1\}$.

To proceed with the proof, it is convenient to define a few more quantities. For a set of tasks $T\subset\{1,\ldots,n\}$ and an assignment function A, *the actual workload of T under A* (denoted $E(A,T)$) is $\Sigma_{t\in T}\mu(t,A(t))$. The *minimal workload for T* (denoted $E_0(T)$) is $\Sigma_{t\in T}b(t)$.

In order to evaluate $f(A)/f(B)$, it is convenient to consider a related task system, μ'. This task system has almost identical entries to μ, differing only in the entries for tasks in $K-L$. In addition, there is a starting time function, s', associated with A such that s' is consistent with (1), (2), and (3), and s' starts all of L, before it starts any of $K-L$.

Lemma 1. Let μ be an $n\times m$ task system, A an assignment function for μ obtained from Algorithm 1, and s the associated starting time function. Let K and L as above. Let B be an optimal assignment for μ. Then there is an $n\times m$ task system μ', and a starting time function s' (with finishing time f') associated with A such that:

(i) The starting time function s' with the assignment A is consistent with (1), (2), and (3) for μ'.

(ii) $f(A)=f'(A)$ and $f(B)=f'(B)$.

(iii) The sets K and L are identical when defined in terms of μ, s, and A, as when they are defined in terms of μ', s', and A'.

(iv) For every $t_1 \in L$ and $t_2 \in K-L$, $s'(t_1)<s'(t_2)$.

(v) $E_0(L)=f'(L)$

Proof. Define μ' as follows. For $t \notin K-L$ $\mu'(t,p)=\mu(t,p)$. For $t \in K-L$, $\mu'(t,A(t))=\mu(t,A(t))$, $\mu'(t,B(t))=\mu(t,B(t))$, and $\mu'(t,p)=\infty$ for $p \neq A(t)$ and $p \neq B(t)$. Note that $f(A)=f'(A)$ and $f(B)=f'(B)$. (In fact B is still an optimal assignment). The starting time function s' is defined to be only slightly different from s. For $t \notin K$, $s'(t)=s(t)$. For $t \in L$:

$$s'(t)= \sum_{u \in L : s(u)<s(t)} \mu'(u,1)$$

For $t \in K-L$:

$$s'(t)=E_0(L) + \sum_{u \in K-L : s(u)<s(t)} \mu'(u,1)$$

Thus, all of the tasks in L are scheduled before all of the tasks in $K-L$, and within each set (L and $K-L$), the order that the tasks are scheduled is the same. Also $E_0(L)=f'(L)$ and the sets K and L are unchanged.

Note that A with s' is consistent with the (1), (2), and (3) for μ'. Condition (1) follows immediately since tasks are executed on the same processors, with the same efficiency as in s. Condition (2) needs to be verified only for tasks in K. If $u \in K$ and $s'(t)>s'(u)$ then since $ef(u,1)=1$,

$ef(t,1) \leq ef(u,1)$. If $t \in L$, $s'(t) > s'(u)$ and $A(u) \neq 1$, then since $s(t) \geq s'(t) > s'(u) = s(u)$, $ef(t,A(u)) \leq ef(u,A(u))$ (using the fact that (2) was true for s). If, $t \in K-L$ and $s'(t) > s'(u)$, condition (2) follows from the fact that in the primed system all tasks $t \in K-L$ have efficiency less than $1/\sqrt{m}$ for all but the first processor. Condition (3) follows in a similar manner. □

This modification of μ to μ' is an *ad hoc* modification needed to make the technical assumption (v).

To obtain a bound on $f(A)/f(B)$ we compute a bound on the ratio using the starting function s' on the task system μ'. Note that $f(K) \geq s(z)$ by condition (3) on A and s. Also, $f(A) = s(z) + E(A,\{z\})$. Therefore, $f(A) \leq E(A,\{z\}) + E(A,K-L) + E(A,L)$. A bound will be obtained on the three summands in terms of $f(B)$. This will be used to get a bound on $f(A)/f(B)$. In the sequel, we will drop the primes of μ, s, and f, for notational convenience. The proof however refers to this modified task system and starting time function.

Lemma 2. Let A,B as above and $1 \leq t \leq n$. Then $E(A,\{t\}) \leq \sqrt{m}\, f(B)$.

Proof. This follows from the fact that A assigns tasks to processors that are somewhat efficient for the task. That is, $f(B) \geq E(B,\{t\}) = \mu(t,B(t)) \geq b(t) = ef(t,A(t))\mu(t,A(t))$. By condition (1), $ef(t,A(t)) \geq 1/\sqrt{m}$. Thus $ef(t,A(t))\mu(t,A(t)) \geq \mu(t,A(t))/\sqrt{m} = E(A,\{t\})/\sqrt{m}$. □

Lemmas 3,4, and 5 are technical facts that are needed for the proof of Lemma 6.

Lemma 3. Let A,B,K,L as above. Then $\sqrt{m}\, E(A,K-L) \leq E(B,K-L)$.

Proof. $E(B,K-L)=\Sigma_{t\in K-L}\, \mu(t,B(t)) \geq \Sigma_{t\in K-L} \sqrt{m}\, \mu(t,A(t))$ by the definition of $K-L$. Thus $E(B,K-L) \geq E(A,K-L)\sqrt{m}$. □

Let $t_i \in L$ be the i^{th} task of L (ordered by starting time). Assume $|L|=q$.

Lemma 4. Let A,L as above. Then
$\Sigma_{t\in L}\, b(t)f(t)=\Sigma_{i=1}^{q} b(t_i)\Sigma_{j=1}^{i} b(t_j) \geq (\Sigma_{i=1}^{q} b(t_i))^2/2=E(A,L)^2/2$.

Proof. The first equality follows from the fact that for $t\in L$, $A(t)=1$, and the fact that all tasks in L are executed sequentially with no intervening tasks. The last equality follows from the definition of $E(A,L)$. To get the inequality $\Sigma_{i=1}^{q} b(t_i)\Sigma_{j=1}^{i} b(t_j) \geq (\Sigma_{i=1}^{q} b(t_i))^2/2$ note that on the left hand side of the inequality the term $b(t_i)b(t_j)$ appears exactly once. This term appears twice on the right hand side for $i\neq j$ and once for $i=j$ in the expansion for $E(A,L)^2$. Dividing by two proves the result. □

Lemma 5. Let $A,B,A^{-1}(\{2,\ldots,m\}),L$ as above. Then

$$E_0(L\cup A^{-1}(\{2,\ldots,m\})) \geq (E(A,L)^2/2f(B))-E(A,L).$$

Proof. The main intuitive idea will be to show that if the set L is large, then $E_0(A^{-1}(\{2,\ldots,m\}))$ must be large. The reason is as follows. Consider a task $t\in L$. Since processor $B(t)$ is somewhat efficient on t, processor $B(t)$ is a candidate processor for t to be assigned to in A. The only reason that t would not be assigned to processor $B(t)$ is because the workload in A of tasks assigned to processor $B(t)$ exceeded $s(t)$ (by condition (3)). Thus if L is a large set, it must be that the total workload of $A^{-1}(\{2,\ldots,m\})$ is large, or else tasks of L would be assigned earlier to different processors. Thus a relation is derived relating the size of L to the value of $E_0(A^{-1}(\{2,\ldots,m\}))$.

To make the above intuitive ideas precise, it is convenient to divide L into m portions. $L_p = L \cap B^{-1}(p)$ is the subset of L that is assigned in B to the pth processor, and in particular is executable efficiently on the pth processor. The set $A^{-1}(p)$ (the tasks assigned to the pth processor by A) is the set that we will show is large, based on the size of L_p. Note that $A^{-1}(\{2,\ldots,m\}) = \cup_{p=2}^{m} A^{-1}(p)$.

For the rest of the proof, fix a value of p and define the following parameters (which have an implicit dependence on p). Let e be the number of tasks in L_p, and let these tasks be denoted $t_1,\ldots,t_e$ with $s(t_1) < s(t_2) < \ldots < s(t_e)$. For $i=1,\ldots,e$, define $U_i = \{t \in A^{-1}(p) : s(t) < s(t_i)\}$. The set U_i is the subset of $A^{-1}(p)$ started before task t_i.

Note that $ef(t_i,p) \geq 1/\sqrt{m}$ by the definition of L. Thus, by condition (3) $f(A^{-1}(p)) \geq s(t_i)$ for every i. Thus some task in U_i finishes after $s(t_i)$ and thus $f(U_i) \geq s(t_i)$.

For $i=1,\ldots,e$, define ef_i to be the smallest efficiency of any of the tasks in U_i on p, i.e., $ef_i = \min_{t \in U_i} ef(t,p)$. Note that if $k > i$, then $ef_k \leq ef_i$ since $U_i \subset U_k$. Also, $ef(t_i,p) \leq ef_i$ by condition (2) since otherwise t_i would have been assigned to p before the last task in U_i was assigned to p. Note that

(1) $f(B)\geq\Sigma_{i=1}^{e}\mu(t_i,p)$.

This follows because $f(B)$ exceeds the actual workload assigned to any one processor in B. Also note that

(2) $\mu(t_i,1)=b(t_i)=ef(t_i,p)\mu(t_i,p)\leq ef_i\mu(t_i,p)$.

Using the above definitions and inequalities, we may now show that the minimal workload of the set $A^{-1}(p)$ must be large if L_p is large.

For convenience define $U_0=\emptyset$, $f(\emptyset)=0$, and $ef_{e+1}=0$. From the definitions we have

(3) $E_0(A^{-1}(p))\geq E_0(U_e)=\Sigma_{t\in U_e}b(t)=\Sigma_{i=1}^{e}\Sigma_{t\in U_i-U_{i-1}}b(t)=\Sigma_{i=1}^{e}\Sigma_{t\in U_i-U_{i-1}}ef(t,p)\mu(t,p)$.

Using (3) and $ef(t,p)\geq ef_i$ for $t\in U_i-U_{i-1}$ produces

(4) $E_0(A^{-1}(p))\geq\Sigma_{i=1}^{e}\Sigma_{t\in U_i-U_{i-1}}ef_i\mu(t,p)$.

Combining (4) with the fact that $\Sigma_{t\in U_i-U_{i-1}}\mu(t,p)=f(U_i)-f(U_{i-1})$ produces

(5) $E_0(A^{-1}(p))\geq\Sigma_{i=1}^{e}ef_i(f(U_i)-f(U_{i-1}))=\Sigma_{i=1}^{e}f(U_i)(ef_i-ef_{i+1})$.

The last equality is obtained by rearranging indices. Now, using $f(U_i)\geq s(t_i)=(f(t_i)-b(t_i))$ we have:

(6) $E_0(A^{-1}(p))\geq\Sigma_{i=1}^{e}(f(t_i)-b(t_i))(ef_i-ef_{i+1})$

To get (6) from (5) note $ef_i\geq ef_{i+1}$.

Let $X=\Sigma_{i=1}^{e}f(t_i)(ef_i-ef_{i+1})$ and $Y=\Sigma_{i=1}^{e}(ef_i-ef_{i+1})b(t_i)$. Then $E_0(A^{-1}(p))\geq X-Y$. Note that $ef_i\leq 1$ for every i. Thus $Y\leq\Sigma_{i=1}^{e}b(T_i)$.

Also, for each k we have:

(7) $ef_kf(t_k)\leq(\Sigma_{i=1}^{k-1}(ef_i-ef_{i+1})f(t_i))+ef_kf(t_k)+(\Sigma_{i=k+1}^{e}ef_i(f(t_i)-f(t_{i-1})))=X.$

Equation (7) follows from $ef_i\geq ef_{i+1}$ for every i and $f(t_i)>f(t_{i-1})$ for every i. Using (2), (7), and (1) (successively) provides:

(8) $\Sigma_{i=1}^{e}b(t_i)f(t_i)\ \leq\ \Sigma_{i=1}^{e}\mu(t_i,p)ef_i\,f(t_i)\ \leq\Sigma_{i=1}^{e}\mu(t_i,p)X\leq f(B)X.$

Thus $X\geq\Sigma_{t\in L_p}f(t)b(t)/f(B)$. This together with the bound on Y provides us with

(9) $E_0(A^{-1}(p))\geq\Sigma_{t\in L_p}(f(t)b(t)/f(B))-\Sigma_{t\in L_p}b(t).$

A special computation is done for $E_0(L)$. We claim that

(10) $E_0(L)\geq\Sigma_{t\in L_1}((f(t)b(t)/f(B))-b(t)).$

This follows from $\Sigma_{t\in L_1}((f(t)b(t)/f(B)-b(t))\leq\Sigma_{t\in L_1}f(t)b(t)/f(B)$ $\leq(\max_{t\in L_1}f(t))\Sigma_{t\in L_1}b(t)/f(B)\ \leq\max_{t\in L_1}f(t)=E_0(L)$. The last equality follows from Lemma 1. This is the step for which we had to transform the original task system into μ'.

Finally, we compute $E_0(L \cup A^{-1}(\{2,\dots,m\}))$. $E_0(L \cup A^{-1}(\{2,\dots,m\}))=(\Sigma_{p=2}^{m} E_0(A^{-1}(p)))+E_0(L)$ by definition. By (9) and (10), $E_0(L \cup A^{-1}(\{2,\dots,m\}))\geq(\Sigma_{p=2}^{m}\Sigma_{t\in L_p}((f(t)b(t)/f(B))-b(t)))+(\Sigma_{t\in L_1}((f(t)b(t)/f(B))-b(t)))$. Thus

(11) $\quad E_0(L \cup A^{-1}(\{2,\dots,m\}))\geq\Sigma_{t\in L} f(t)b(t)/f(B)-b(t)$.

By Lemma 4 and equation (11), $E_0(L \cup A^{-1}(\{2,\dots,m\}))\geq(E(A,L)^2/2f(B))-E(A,L)$. □

Lemma 6. Let A,B,K,L as above. Then $E(A,K-L)+E(A,L)\leq(1.5\sqrt{m}+1+(1/2\sqrt{m}))f(B)$.

Proof. Combining Lemmas 3 and 5 we derive that $E(B,\{1,\dots,n\})\geq E_0(L \cup A^{-1}(\{2,\dots,m\}))+E(B,K-L)\geq(E(A,L)^2/2f(B))-E(A,L)+\sqrt{m}\,E(A,K-L)$. Note that $mf(B)\geq E(B,\{1,\dots,n\})$, since B can do no better than divide the workload of all the tasks equally among the m processors. Thus $mf(B)\geq(E(A,L)^2/2f(B))-E(A,L)+\sqrt{m}\,E(A,K-L)$. Let $a=E(A,K-L)/f(B)$ and $b=E(A,L)/f(B)$. Then, $2m\geq b^2-2b+2a\sqrt{m}$. To prove the lemma, we determine the maximum value for $(E(A,L)+E(A,K-L))/f(B)=a+b$ subject to $2m\geq b^2-2b+2a\sqrt{m}$.

Note that the maximum value of $a+b$ occurs at $a=(2m-b^2+2b)/2\sqrt{m}$ (for any fixed value of b). Now, to maximize $b+((2m-b^2+2b)/2\sqrt{m})$, differentiate with respect to b, and set the derivative to 0. Solving $1+(2-2b)/2\sqrt{m}=0$ produces $b=1+\sqrt{m}$. For that value of b, the maximum value for a is $(\sqrt{m}/2)+(1/2\sqrt{m})$. Thus the maximum value of $a+b$ is $1.5\sqrt{m}+1+(1/2\sqrt{m})$ and the lemma is proved. □

Theorem 1. Let A be an assignment function for a task system consistent with (1), (2), and (3). Let B be an optimal assignment function. Then $f(A)/f(B) \leq 2.5\sqrt{m}+1+(1/2\sqrt{m})$.

Proof. Reduce μ to μ' as above, and analyze $f(A)/f(B)$, by analyzing the assignment function A together with the starting function s'. Note that by Lemma 2 $E(A,\{z\}) \leq \sqrt{m}\, f(B)$. Use $f(A) \leq E(A,\{z\})+E(A,K-L)+E(A,L)$ together with $E(A,\{z\}) \leq \sqrt{m}\, f(B)$ and $E(A,K-L)+E(A,L) \leq (1.5\sqrt{m}+1+(1/2\sqrt{m}))f(B)$ to get the theorem. □

4. Achieving the bound to a constant factor

Consider the following $m+1 \times m$ task system. For $t=1,\ldots,m-2$, $\mu(t,1)=1$, $\mu(t,t+2)=\epsilon+\sqrt{m}$, and $\mu(t,p)=\infty$ for $p\neq 1,t+2$. The value $\epsilon>0$ is a parameter which will be sent to zero to get as tight a bound as possible. In addition, $\mu(m-1,2)=\epsilon$ and $\mu(m-1,p)=\infty$ for $p\neq 2$; $\mu(m,3)=m-2-\epsilon$, $\mu(m,2)=\sqrt{m}$, and $\mu(m,p)=\infty$ for $p\neq 2,3$; $\mu(m+1,1)=\sqrt{m}$, $\mu(m+1,3)=m$, and $\mu(m+1,p)=\infty$ for $p\neq 1,3$.

An optimal assignment is as follows. For $t=1,\ldots,m-2$, $B(t)=t+2$, requiring time $\epsilon+\sqrt{m}$. $B(m-1)=B(m)=2$, and thus the actual workload of the tasks assigned to processor 2 is $\epsilon+\sqrt{m}$. Finally, $B(m+1)=1$, requiring time $\sqrt{m}$.

An assignment consistent with (1), (2), and (3) may proceed as follows. First note that no tasks may be assigned to processors $4,\ldots,m$ since no task is sufficiently efficient on those processors. $A(t)=1$ for $t=1,\ldots,m-2$ with $s(t)=t-1$. $A(m-1)=2$ with $s(m-1)=0$. $A(m)=A(m+1)=3$ with $s(m)=0$ and $s(m+1)=m-2-\epsilon$. It is straightforward to verify that this satisfies (1), (2), and (3).

Due to the tasks assigned to the third processor, $f(A)=2m-2-\epsilon$. The ratio between finishing times is $(2m-2-\epsilon)/(\epsilon+\sqrt{m})$. As ϵ gets smaller the ratio approaches $2\sqrt{m}-(2/\sqrt{m})$. □

5. Scheduling with limited information

In this section we indicate that no algorithm with certain limitations can have better worst case behavior which is orders of magnitude better than that of Algorithm 1. Specifically, assume that the scheduler knew the matrix of efficiencies but not the matrix that specifies the time requirements. Such a circumstance is imaginable if the tasks had known characteristics, but the actual time required was unknown (for example due to loops of undetermined number of iterations).

If only the efficiencies are known, any algorithm that develops an assignment based only on this information must be at least $\sqrt{m}$ times worse than optimal in the worst case. This is true even if before assigning a task to start at a given time, the scheduler knows the absolute time requirement of all tasks finished by that given time (as one might expect that the scheduler would at least have access to that information). This indicates that our algorithm is within a constant factor of optimal for this type of scheduling. This fact is independent of running time; no algorithm can do better than $\sqrt{m}$ times worse than optimal irrespective of the amount of time it takes.

Consider the following $m \times m$ efficiency matrix. The entries are $ef(t,1)=1$, and $ef(t,p)=1/\sqrt{m}$ for $p>1$. If an algorithm assigns all of the m tasks to the first processor then in the worst case, the assignment is $\sqrt{m}$ times worse than optimal. This occurs when $\mu(t,1)$ is the same for $t=1,\ldots,m$. Then assigning all tasks to the first processor requires time m, whereas assigning the tth task to the tth processor requires time $\sqrt{m}$.

On the other hand, if any of the tasks are not assigned to the first processor, the performance ratio is still $\sqrt{m}$ times worse than optimal in the worst case. Assume that the tth task is the first one not assigned to processor 1, and is instead assigned to processor p. Assume that $\mu(u,1)=\epsilon$ for $u \neq t$ and $\mu(t,1)=1$. Then an asymptotically optimal assignment has $B(t)=1$ and $B(u)=u$ for $u \neq t$. This requires time $1+\epsilon$, whereas the heuristic requires at least time $\sqrt{m}$ by assigning the t^{th} task to the t^{th} processor. Thus the heuristic is $\sqrt{m}$ times worse than optimal. □

Note that the processors are "uniform" in this example. That is, $ef(t,p)$ depends only on p and is independent of t. For this case, an algorithm was presented in [7] which is asymptotic to $\sqrt{m}$ times worse than optimal in the worst case, even if there are precedence constraints. In this environment of incomplete information, the heuristic presented in [7] is asymptotically optimal.

Even in this restricted environment, where the matrix μ is not known, the above result is not a strong result for the following reason. A very weak form of preemption avoids the difficulties illustrated in the example. Specifically, if at "run-time", a task which has been begun on one processor may be reassigned to be rerun from the beginning on another, then the above task system may be executed relatively efficiently even if the absolute time requirements are not known a priori.

6. Two improved algorithms

In this section we consider two minor modifications to Algorithm 1 which provide slightly better performance bounds:

Algorithm 2.

Step 1. Same as Step 1 in Algorithm 1, except that if $ef(t,p)=ef(u,p)$, then

t is ordered before u if $\mu(t,p)\geq\mu(u,p)$.

Steps 2-4. Same as Algorithm 1.

This algorithm satisfies the following stronger form of condition (2).

(2′) If $s(t)>s(u)$ then either $ef(t,A(u))<ef(u,A(u))$ or $ef(t,A(u))=ef(u,A(u))$ and $\mu(t,A(u))\leq\mu(u,A(u))$.

We obtain a bound on this algorithm of approximately $(1+\sqrt{2})\sqrt{m}$ times worse than optimal.

Using the notation of Section 3, we wish to place a bound on $(E(A,K-L)+E(A,\{z\}))/f(B)$. If $K-L$ is empty then $E(A,K-L)+E(A,\{z\})/f(B)=E(A,\{z\})/f(B)\leq\sqrt{m}$ by Lemma 2. If $K-L$ is not empty, choose $t\in K-L$. Then $f(B)\geq\mu(t,B(t))\geq\sqrt{m}\,\mu(t,1)\geq\sqrt{m}\,\mu(z,1)\geq\mu(z,A(z))=E(A,\{z\})$. Thus $E(A,\{z\})/f(B)=1$. Furthermore, by Lemma 3, $\sqrt{m}\,E(A,K-L)\leq E(B,K-L)$. Since $mf(B)\geq E(B,K-L)$, we conclude $E(A,K-L)/f(B)\leq\sqrt{m}$. Hence in either case $(E(A,K-L)+E(A,\{z\}))/f(B)\leq\sqrt{m}+1$.

Applying the inequality $E_0(L\cup A^{-1}(\{2,\ldots,m\}))\leq mf(B)$ to Lemma 5, we obtain $mf(B)\geq(E(A,L)^2/2f(B))-E(A,L)$. Let $x=E(A,L)/f(B)$. Then $m\geq(x^2/2)-x$. Solving this for the maximum possible value for x yields $x=(1+\sqrt{2m+1})$. Thus $E(A,L)/f(B)\leq 1+\sqrt{2}\sqrt{m}+(1/2\sqrt{2}\sqrt{m})$. Using the above inequalities proves:

Theorem 2. Let A be an assignment given by Algorithm 2 and let B be an optimal assignment. Then $f(A)/f(B)\leq(1+\sqrt{2})\sqrt{m}+2+(1/\sqrt{8}\sqrt{m})$.

Next we present a different modification of Algorithm 1, which has a worst case performance ratio of $1.5\sqrt{m}$ time worse than optimal. This algorithm has running time $O(m^m+mn\log(n))$, and is therefore quite useless if the number of processors is an input to the problem. However, in the situation that the scheduler needs to deal with a bounded number of processors, this is still an $O(n\log(n))$ algorithm.

Algorithm 3.

1. Devise an assignment A_1 and starting function s, using Algorithm 1.

2. Let u be the task that has latest starting time in s, and let $U=\{t:f(t)>s(u)\}$.

3. Determine an optimal assignment for the set U (considered as entire task system) by trying all possible assignments. Assign each task in U to the processor that it is assigned to in this optimal schedule (the tasks in U are assigned to start after the other tasks already assigned to the relevant processors).

First note that $|U|\leq m$. This follows from the fact that if $t_1,t_2 \in U$ then $A_1(t_1)\neq A_1(t_2)$. For if two tasks are assigned to the same processor, the later one must have starting time which is later than $s(u)$. Note that the running time of Algorithm 3 is $O(m^m+mn\log(n))$. Step 1 requires time $O(mn\log(n))$. Step 3 requires time $O(m^m)$ since each possible assignment must be considered. Once these m^m schedules have been tried, determining the best assignment also requires no more time than $O(m^m)$. We proceed as in Section 3, letting A be an assignment resulting from Algorithm 3, and letting B be an optimal assignment for μ.

The sets K and L are defined on the basis of u. At time $s(u)$, if $ef(u,p)=1$, then (informally speaking) processor p is still executing tasks that have efficiency one. Assume again that $p=1$. The set K consists of all tasks assigned to processor one with starting time earlier than $s(u)$. As in Section 3, $L=\{t\in K: ef(t,B(t))\geq 1/\sqrt{m}\}$. The proof that $E(A,L)+E(A,K-L)\leq(1.5\sqrt{m}+1+(1/2\sqrt{m}))f(B)$ still applies, even though the sets L and $K-L$ are not the same as those defined in Section 3.

Let $f^*(U)$ be the "optimal finishing time" of the set of tasks U when considered as a separate task system (i.e., not as part of μ). Then it follows from the design of the algorithm that $f(A)\leq E(A,K-L)+E(A,L)+f^*(U)$. Thus to obtain a bound on $f(A)/f(B)$ it suffices to get a bound on $f^*(U)$ in terms of $f(B)$. But clearly $f^*(U)\leq f(B)$. Using this fact together with Lemma 6 provides:

Theorem 3. Let A be a schedule obtained with Algorithm 3, and let B be an optimal schedule. Then $f(A)/f(B)\leq(1.5\sqrt{m}+2+(1/2\sqrt{m}))$.

To show that the bounds on Algorithms 2 and 3 are achievable (to within a constant factor), consider the $m \times m$ task system given by $\mu(t,1)=1$, $\mu(t,t)=\epsilon+\sqrt{m}$, and $\mu(t,p)=\infty$ for $p\neq 1,t$. The heuristics assign all tasks to the first processor and perform $\sqrt{m}$ times worse than optimal. □

The ϵ approximation algorithms of Horowitz and Sahni [5,12] require time n^{2m}/ϵ. Even for a relatively small value of m this algorithm requires too much time, as n^{10} or n^{20} algorithms are actually not feasible. Algorithm 3 requires time $O(n\log(n))$ for any fixed value of m and thus for most values of m it is more feasible than the algorithms of [5,12].

7. Improving the constant factor.

The threshhold of efficiency chosen by Algorithms 1, 2, and 3 is that if $A(t)=p$ then $ef(t,p)\geq 1/\sqrt{m}$. While we feel that this is the best threshhold to choose, we can actually obtain a better bound using a different threshhold.

Consider the heuristic which is identical to Algorithm 1, except that it uses a threshhold of $1/c\sqrt{m}$. Most of the analysis of the algorithm is the same with the following easily verifiable changes. (Note that the choice of whether tasks go into L or K-L depends on the threshhold of $1/c\sqrt{m}$.) It is curious to note that Lemmas 4 and 5 do not depend at all on the threshhold.

(1) $E(A,\{z\}) \leq c\sqrt{m}\, f(B)$.

(2) $c\sqrt{m}\, E(A,K\text{-}L) \leq E(B,K\text{-}L)$

(3) In the proof of Lemma 6, when trying to maximize $a+b$, the relevant inequality that constrains the maximization is $a \leq (2m-b^2+2b)/2c\sqrt{m}$. This occurs at $b=1+c\sqrt{m}$. For that value of b, the maximum value of a is $(\sqrt{m}(2-c^2)/2c)+(1/2c\sqrt{m})$.

(4) Putting together (1) and (3) provides a bound on the algorithm of $\sqrt{m}((3c/2)+(1/c))$, neglecting lower order terms. The smallest value of this occurs at $c=\sqrt{2/3}$, where the bound is $\sqrt{6m}$ times worse than optimal. This is marginally better than the original bound of $\sqrt{6.25m}$ times worse than optimal. □

Using this technique in conjunction with Algorithm 2 provides slightly better results. If K-L is empty then $(E(A,\{z\})+E(A,K\text{-}L))/f(B) \leq c\sqrt{m}$ and $E(A,L)/f(B) \leq \sqrt{2}\sqrt{m}$. If K-L is nonempty then $E(A,A^{-1}(\{2,\ldots,m\}))/f(B) \leq 1$, and $E(A,K\text{-}L)+E(A,L) \leq ((c/2)+(1/c))\sqrt{m}$. The best value of c to choose is the one that minimizes $\max(c+\sqrt{2},(c/2)+(1/c))$. Choosing $c=2-\sqrt{2}$ provides a bound of $2\sqrt{m}$. □

Using this technique in conjunction with Algorithm 3 provides marginally better results. In that case, $E(A,\{z\})$ does not effect the bound, and $E(A,K\text{-}L)+E(A,L) \leq \sqrt{m}\,((c/2)+(1/c))$ plus lower order terms. For this case, choosing $c=\sqrt{2}$ gives a bound on Algorithm 3 of being at most $\sqrt{2}\sqrt{m}$ times worse than optimal. □

8. Two other heuristics

The following is "Algorithm D" from [6]. This is the algorithm that was shown to be between 2 and m times worse than optimal in the original paper of Ibarra and Kim, and we will show that in the worst case it is at least $1+\log_2(m)$ times worse than optimal.

Algorithm D:

Step 1. $sum_p=0$ for $1\leq p\leq m$, $S=\{1,...,n\}$.

Step 2. If $S=\emptyset$ then end.

Step 3. Find an index $t\in S$ such that $\min_p \{sum_p+\mu(t,p)\}\leq\min_p \{sum_p+\mu(t',p)\}$ for all $t'\in S$. Let p be such that $sum_p+\mu(t,p)$ is minimum. Define $A(t)=p$. Set $sum_p=sum_p+\mu(t,p)$ and $S=S-\{t\}$. Go to step 2.

The basic idea behind "Algorithm D" is the following. After i tasks have been scheduled, the scheduler sets up a temporary goal of trying to schedule one more task and minimize the total finishing time of the $i+1$ task, task system. The scheduler chooses the task to use as the $i+1^{st}$, and which processor to assign it to. After iterating this procedure n times, all tasks have been assigned.

The following is an $m \times m$ task system for which algorithm D performs poorly. The entries are $\mu(t,p)=1$ if $m-t+1\geq p$ and $\mu(t,p)=\infty$ if $m-t+1<p$. An optimal schedule has $B(t)=m-t+1$ with a finishing time of 1. However, the following schedule is consistent with Algorithm D.

Assume that m is a power of 2. The first $m/2$ tasks to be assigned are tasks $1,\ldots,m/2$ with $A(t)=t$. After these are assigned, the workload assigned to each of the first $m/2$ processors is 1. Next, tasks $(m/2)+1,\ldots,(3m/4)$ are assigned. Task $(m/2)+p$ is assigned to processor p. Continuing in this manner, tasks $1,(m/2)+1,(3m/4)+1,\ldots$ are all assigned to processor 1. Thus the total workload assigned to processor 1 is $1+\log_2(m)$. The finishing time of A is $1+\log_2(m)$. $\square$

The exact worst case performance of this algorithm is left as an open problem. Note, that while it may be a better algorithm than the algorithms of this paper in terms of worst case performance, it has a longer running time. Algorithm 1 requires time $O(mn\log(n))$ whereas "Algorithm D" requires time $O(mn^2)$.

We mention one final trivial algorithm which is quite simple, and in fact generalizes to the situation where there is a precedence relation. If there is a precedence relation between tasks, then the starting time function is needed in order to determine the finishing time of an assignment function. This is due to the fact that if $t<u$, the restriction $s(u)\geq f(t)$ is imposed. Thus different starting functions for the same assignment might have different finishing times. In particular, the finishing time is not just the maximum of the workloads of each processor. Instead, the finishing time of the task system is the maximum finishing time of the individual tasks. Other than that, none of the definitions change, and the concepts of optimal schedule and performance ratio still apply.

This naive algorithm is to assign each task to a processor which has efficiency one for the task. Since there is a precedence relation in this general case, processors may be temporarily idle and reactivated. However, we insist that at no time (before the finishing time) are all processors idle.

Let A be a schedule consistent with the heuristic and B an optimal schedule for such an ordered set of tasks. Clearly $f(A) \leq E_0(\{1,...,n\})$, since $E(A,\{1,...,n\})=E_0(\{1,...,n\})$, and at least one unit of $E(A,\{1,...,n\})$ is executed per unit time (before the finishing time). However, $f(B) \geq E_0(\{1,...,n\})/m$. This follows from the fact that B can certainly do no better than assign all tasks to their best processor, and furthermore do them m at a time. □

This bound of m times worse than optimal is achievable even without any precedence constraint. Consider the $m \times m$ task system with $\mu(t,1)=1$ (for $t>1$), $\mu(t,t)=1+\epsilon$, and $\mu(t,p)=\infty$ for $p \neq 1,t$. The heuristic assigns all m tasks to processor 1 and requires time m. Optimal scheduling assigns task t to processor t and requires time $1+\epsilon$. As ϵ goes to 0, the ratio approaches m. □

This algorithm was worth mentioning only because precedence constraints have a tendency to make scheduling heuristics perform quite poorly. For example, it can be shown that the natural extension of Algorithm 1 to the case that there are precedence constraints provides an algorithm which is as bad as $m\sqrt{m}$ times worse than optimal in the worst case.

9. Conclusions

We have presented a number of algorithms with behavior which is $O(\sqrt{m})$ times worse than optimal in the worst case. While this is the best possible behavior for a limited class of schedules (as explained in Section 5), there is little reason to believe that this cannot be improved when considering a larger class of schedules. Algorithm *D* of [6] still has not been fully classified, and appears to be a promising candidate for better order of magnitude behavior.

An even more difficult problem seems to be finding an algorithm that has better than $O(m)$ behavior in the presence of a precedence constraint. Such algorithms have been devised in the situation that the processors are of different speeds, but uniformly so [7] (i.e., the efficiency of a processor on a task is independent of the task).

Acknowledgements

The authors would like to express their thanks to Albert Meyer for many helpful comments on earlier drafts of this paper.

References

1. J. Bruno, E. G. Coffman Jr., and R. Sethi, Scheduling independent tasks to reduce mean finishing time, *CACM 17*, 7 (July 1974), 382-387.

2. J. Bruno, E. G. Coffman Jr., and R. Sethi, Algorithms for minimizing mean flow time, *IFIP 74*, North-Holland, Amsterdam, pp. 504-510.

3. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco (1979).

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

5. E. Horowitz and S. Sahni, Exact and approximate algorithms for scheduling nonidentical processors, *JACM, 23*, 2 (April 1976), 317-327.

6. O. H. Ibarra, and C. E. Kim, Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors, *JACM 24*, 2, (April 1977), 280-289.

7. J. M. Jaffe, Efficient Scheduling of Tasks Without Full Use of Processor

Resources, MIT Laboratory for Computer Science Technical Memo 122, January 1979. Also to appear in *Theoretical Computer Science.*

8. J. M. Jaffe, Bounds on the Scheduling of Typed Task Systems, MIT, Laboratory for Computer Science Technical Memo 111, September 1978. Also, submitted to *SIAM J. Comput.*

9. D. G. Kafura and V. Y. Shen, Task scheduling on a multiprocessor system with independent memories, *SIAM J. Comput.* 6, (March 1977), 167-187.

10. E. L. Lawler and J. Labetoulle, On preemptive scheduling of unrelated parallel processors by linear programming, *JACM*, *25*, 4, 1978 612-619.

11. J. W. S. Liu and C. L. Liu, Performance Analysis of Multiprocessor Systems Containing Functionally Dedicated Processors, *Acta Informatica*, *10*, 1, (1978) 95-104.

12. S. Sahni, Algorithms for scheduling independent tasks, *JACM*, *23*, 1, (Nov. 1976), pp 116-127.

13. S. Sahni and T. Gonzalez, Preemptive scheduling of two unrelated machines, Tech. Rep. 76-16 Comptr. Sci. Dept., U. of Minnesota, Minneapolis, MN (November 1976).