appeared in the International Conference on Parallel Processing, August 1990

AN APPROXIMATE ALGORITHM FOR THE PARTITIONABLE INDEPENDENT TASK SCHEDULING PROBLEM

Krishna P. Belkhale Prithviraj Banerjee

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL-61801

ABSTRACT

Scheduling a collection of tasks on a multiprocessor, consisting of p processors, that minimizes the maximum completion time has attracted a lot of attention in the literature [12]. In this paper, we introduce a new problem of scheduling a collection of independent tasks on a multiprocessor, called the *partitionable independent task scheduling* problem. Associated with each task, we are given the time it takes to run on a uniprocessor, and speedup that can be obtained by running it on i processors, $1 \le i \le p$. We present an *approximate* algorithm that guarantees a solution within $\frac{2}{(1+1/p)}$ of the optimal solution, under a reasonable assumption on the speedup functions.

1.INTRODUCTION

The problem that we are interested will be called the *partitionable independent task scheduling* problem. This problem arises when a parallel algorithm is designed for the circuit extraction problem, where the input is specified hierarchically [3]. It involves scheduling n independent tasks $T_1, \cdots T_n$ on p processors. The time that each task T_i takes to run on a uniprocessor is assumed to be known *a priori*, and is denoted by $\tau(T_i)$. The additional freedom, that has not been considered in previous formulations, involves the option of running a task on multiple processors. Associated with each task T_i, we assume that we are given the estimated speedup $\sigma_k(T_i)$ that can be obtained by running T_i on k processors, $1 \le k \le p$. In other words, the task T_i completes in time $\frac{\tau(T_i)}{\sigma_k(T_i)}$ when run on k processors.

Let $\eta_j(T_i) = \frac{\sigma_j(T_i)}{j}$. We assume that $\eta_j(T_i) \ge \eta_{j+1}(T_i)$, for all i, j, where $1 \le i \le n$ and $1 \le j \le p-1$. This assumption is called the *efficiency* assumption. It is a reasonable assumption, as it can be easily proved that if the speedup curve of a task has a convex shape, then it satisfies the efficiency assumption, though the converse is not true. In addition to the *efficiency assumption* we assume $\sigma_j(T_i) \le \sigma_{j+1}(T_i)$, for all i, j, where $1 \le i \le n$ and $1 \le j \le p-1$, without any loss of generality.

* This research was supported in part by the National Science Foundation Award under Grant NSF MIP 87-14646, and in part by an equipment grant NSF CCR 87-05240.

A schedule S is defined as a specification of tasks to processors, i.e it assigns an ordered list of tasks for each of the processors. The combined ordering of the tasks in a schedule should define a partial order, i.e should be acyclic. Some of the tasks could be assigned to more than one processor. A task T_i assigned to a set of processors can start only after all the processors in the set have finished the task assigned to them immediately before T_i. For a schedule S, let $f(S)$ denote the final finish time. The objective of the problem is to find the schedule S which minimizes $f(S)$.

2. PREVIOUS WORK

The problem described in this paper has been considered before, for the special case when a task is restricted to run on a single processor. This problem is called the *independent task scheduling problem.* Even, for this special case, the problem is NP-hard for all $p \ge 2$. In this section, we review an algorithm, called the *LPT* algorithm [4]. This algorithm generates a solution that is provably within $\frac{4}{3} - \frac{1}{3p}$ of the optimal [4].

The *LPT* algorithm is one among a class of algorithms called the *list scheduling* algorithms. The basic idea in these algorithms is to consider tasks in a specific order from a list, and build a feasible schedule. At any stage we have a partial schedule. Let F_j denote the current finish time of processor j. The complete algorithm is described below.

procedure *list_schedule*

[1] Set i to 1, and F_j to 0 for all $1 \le j \le p$.

[2] Choose a processor j with the smallest F_j among all processors (i.e the processor with the current minimum finish time).

[3] Assign T_i as the next task to processor j and update F_j to $F_j + \tau(T_i)$.

[4] If $i \ne n$ increment i and go to 2. In the other case the algorithm terminates with overall schedule value equal to the maximum of all the F_j values.

end *list_schedule.*

The *LPT* (largest processing time) algorithm is a *list scheduling* algorithm that uses a list of tasks that is sorted in decreasing order based on execution times. The *LPT* algorithm can be implemented in time $O(n \log n + n \log p)$.

3. APPROXIMATE ALGORITHM

The *partitionable independent task scheduling* problem allows the possibility of executing a task among multiple processors. The problem continues to be NP-hard. The main idea in the approximate algorithm is to construct an *LPT* schedule and iteratively modify it by assigning some tasks to more processors, provided this results in a decrease in the schedule time. At any stage, for every task T_i, we have a number s_i which denotes the number of processors that has been allocated to T_i. During the course of the algorithm, the number of processors allocated to a task may be increased iteratively. Each increase is allowed only if it results in an "immediate" decrease in the schedule time.

procedure *part_schedule*

[1] Set s_i to 1 for all $1 \le i \le n$. Also, set a to p (the invariant maintained is $a = p - \sum_{i=1}^{n} s_i$). Set the current schedule C to be an *LPT* schedule.

[2] If $a = 0$ then go to 5, else determine a task T_i such that for all j, $1 \le j \le n$, $\frac{\tau(T_i)}{\sigma_{s_i}(T_i)} \ge \frac{\tau(T_j)}{\sigma_{s_j}(T_j)}$. Let $h = \frac{\tau(T_i)}{\sigma_{s_i}(T_i)}$.

[3] If $a = 1$ and $s_i = 1$ then go to 5, else do the following. If $f(C) \ne h$ then go to 5, else continue.

[4] If $s_i = 1$, set a to $a-2$, else set a to $a-1$. Increment s_i by 1. Construct a schedule C' as follows. The tasks T_j, with $s_j > 1$ are first assigned to the required number of processors (their total requirement is less than p). This is followed by an assignment of the remaining tasks using an *LPT* algorithm. If $f(C') > h$ then decrement s_i and go to 5, else set C to C' and go to 2.

[5] The output schedule A is set to C.

end *part_schedule*.

Example 1: Consider the problem instance shown in Fig. 1(a). The initial solution is shown in Fig. 1(b). Going through the steps, T_1 is selected in Step 2, i.e $i = 1$, and h is set to 10. The check in Step 3 succeeds as the current solution is 10, which is equal to h. The task T_1 is temporarily assigned 2 processors, and the check in Step 4 is performed. As can be seen in Fig. 1(c), all the tasks can be scheduled within time 8, which is not higher than 10. Thus T_1 is currently assigned 2 processors. Next, the task T_2 is selected in Step 2, i.e $i = 2$, and h is set to 8. The check in Step 3 succeeds as the current solution is 8, which is equal to h. Thus T_2 is temporarily assigned two processors, and the check in Step 4 is performed. As can be seen in Fig. 1(d), all the tasks can be scheduled within time 6, which is not higher than 8. Thus T_2 is currently assigned 2 processors. Next, the task T_1 or T_3 could be selected with $h = 5$, and we assume it is the former. However, the current solution is 6, which is not equal to h. The algorithm is hence terminated, with the schedule in Fig. 1(d) as the solution.

Let S be a schedule. We define $k_i(S)$ to denote the number of processors assigned to task T_i in schedule S. A task T_i assigned to $j = k_i(S) > 1$ processors can be looked upon as a collection of j subtasks each having execution time $\frac{\tau(T_i)}{\sigma_j(T_i)}$. These subtasks will be referred to as *split* tasks. The other tasks T_i, with $k_i(S) = 1$, will be referred to as *unsplit* tasks. In scheduling a collection of *split* and *unsplit* tasks, all *split* tasks that are from the same task are required to start together. The current solution, during the course of the *part_schedule* algorithm, is obtained by first assigning the *split* tasks followed by the assignment of the *unsplit* tasks in an *LPT* fashion. The variable a in the algorithm guarantees that there are at most p *split* tasks. Thus all the *split* tasks start at the same time, satisfying the scheduling requirement.

We define $AVG(S)$ to denote $\frac{1}{p}\sum_{i=1}^{n} \frac{\tau(T_i)k_i(S)}{\sigma_{k_i(S)}(T_i)}$. We define $h(S) = \max\{ \frac{\tau(T_i)}{\sigma_{k_i(S)}(T_i)}, 1 \le i \le n \}$. In other words, $h(S)$ is the maximum execution time of a *split* or an *unsplit* task in S.

Lemma 1: $f(S) \ge \max\{ h(S), AVG(S) \}$.
Proof: The finish time cannot be less than the average execution load, and also cannot be less than the execution time of a task in the schedule. Q.E.D.

Lemma 2: If $f(A) \ne f(OPT)$, then $k_i(A) \le k_i(OPT)$, for all $1 \le i \le n$, where A is the schedule of the *part_schedule* algorithm and *OPT* is the optimum schedule.
Proof: Assume $k_i(A) > k_i(OPT)$ for some i, $1 \le i \le n$. Let $l = k_i(A)$ and $m = k_i(OPT)$ for the sake of convenience. Because the number of processors allocated to a task is increased iteratively, there is a stage in the *part_schedule* algorithm when $s_i = m$. The task T_i is assigned one more processor, at some stage in the algorithm, because the condition in Step 3 is true in that stage. This implies that the current schedule at that stage has finish time equal to $\frac{\tau(T_i)}{m}$. Since the current schedule time only improves during the course of the algorithm, we conclude $f(A) \le \frac{\tau(T_i)}{m}$. From Lemma 1, since T_i is assigned to m processors in an optimal solution, we have $f(OPT) \ge \frac{\tau(T_i)}{m}$. This implies $f(A) \le f(OPT)$, which leads to a contradiction, proving the lemma. Q.E.D.

Lemma 3. If $f(A) \ne f(OPT)$, then $AVG(A) \le AVG(OPT)$.
Proof: $AVG(S)$ of a schedule S can be rewritten as $\frac{1}{p}\sum_{i=1}^{n} \frac{\tau(T_i)}{\eta_{k_i(S)}(T_i)}$. Because of the *efficiency assumption*, this implies that if a task T_i is assigned to more processors, then its contribution to the function *AVG* will increase. From Lemma 2, the optimal does not assign lesser number of processors for any task as compared to the *part_schedule* algorithm, proving the lemma. Q.E.D.

Lemma 4: Let S be a schedule that is obtained as in Step 4 of the algorithm from a collection of *split* and *unsplit* tasks, where the number of *split* tasks, x, is no more than p. Let y be the execution time of the $(p-x+1)^{th}$ largest *unsplit* task (if there are not that many *unsplit* tasks y is taken as 0). Then, $f(S) \le \max\{ h(S), AVG(S) + y(1-1/p) \}$.
Proof: The finish time of the schedule S could either be the finish time of a task T_i scheduled at time 0, or of some task which was scheduled later. In the first case, T_i must be a *split*

or an *unsplit* task with the maximum execution time. In other words, $f(S)=h(S)$. In the other case, the task T_i is an *unsplit* task and $\tau(T_i)\le y$. This is because the highest $p-x$ *unsplit* tasks are scheduled to start at time 0. The task T_i starts at time $f(S)-\tau(T_i)$. From the property of the *LPT* algorithm, none of the processors have completed before this time. Thus $f(S)-\tau(T_i)\le AVG'$, where AVG' is average of all the tasks already started at this time. But $AVG' \le AVG(S)-\frac{\tau(T_i)}{p}$. Thus $f(S)\le AVG(S)+\tau(T_i)(1-1/p)\le AVG(S)+y(1-1/p)$, proving the lemma. Q.E.D.

Lemma 5: Let C be a schedule in the *part_schedule* algorithm at some stage. Let z be the execution time of a *split* task in C. Then, $f(C)\le 2z$.

Proof: Let T_i be the task corresponding to z. Thus, $z=\frac{\tau(T_i)}{\sigma_{k_i(C)}(T_i)}$. Clearly, $f(C)\le \frac{\tau(T_i)}{\sigma_{k_i(C)-1}(T_i)}$, as one more processor was allocated to T_i at some stage. Because of the *efficiency assumption* it can be easily proved that $\frac{\tau(T_i)}{\sigma_{k_i(C)-1}(T_i)} \le \frac{2\tau(T_i)}{\sigma_{k_i(C)}(T_i)}$. Thus combining the inequalities, we have $f(C)\le 2z$. Q.E.D.

Lemma 6: Let C be a schedule in the *part_schedule* algorithm at some stage. Let x be the number of *split* tasks and let y be the execution time of the $(p-x+1)^{th}$ largest *unsplit* task. Then

(1) $y \le \frac{\tau(T_i)}{\sigma_{k_i(C)}(T_i)}$, for all i such that $k_i(C)>1$.

(2) $f(C)\le \max\{h(C), \frac{2AVG(C)}{(1+1/p)}\}$.

Proof: For (1), we have to prove y is less than the execution times of all the *split* tasks. Let z be the minimum execution time among all the *split* tasks. Let T_i be the task realizing the minimum, i.e $z=\frac{\tau(T_i)}{\sigma_{k_i(C)}(T_i)}$. From Lemma 5, we have $f(C)\le 2z$. Assuming $z<y$, then we see that $f(C)\ge z+y$. This is because the *unsplit* task corresponding to y will start after the completion of a *split* task with execution time not lesser than z, or an *unsplit* task with execution time not lesser than y. Thus we have $2z \ge z+y$, i.e $z \ge y$, arriving at a contradiction, proving (1).

From (1), the execution time of the *unsplit* task corresponding to y, is less than or equal to the execution time of the x *split* tasks. It also is less than or equal to the execution times of the $p-x$ largest *unsplit* tasks, from the definition of y. Thus $AVG(C)\ge y+y/p$. In other words, $y \le \frac{AVG(C)}{(1+1/p)}$. Substituting this in the result of Lemma 4, we obtain the result of (2). Q.E.D.

Theorem 1: For an instance of the problem, let A be the schedule given by the *part_schedule* algorithm, and let *OPT* be an optimum schedule. Then, $f(A)\le \frac{2f(OPT)}{(1+1/p)}$.

Proof: We assume $f(A)\ne f(OPT)$, or else the theorem is trivially true. There are two cases.

Case 1: $f(A)\ne h(A)$. From Lemma 1, $f(A)\ge \max\{h(A), AVG(A)\}$. Therefore $f(A)>h(A)$. From statement (2) of Lemma 6, this implies $f(A)\le \frac{2AVG(A)}{(1+1/p)}$. From Lemma 3, $AVG(A)\le AVG(OPT)$, and from Lemma 1, $AVG(OPT)\le f(OPT)$. Combining the inequalities, we have $f(A)\le \frac{2f(OPT)}{(1+1/p)}$.

Case 2: $f(A)=h(A)$. The theorem is proved in this case by considering the various possible cases of termination in the *part_schedule* algorithm. Let T_b be a task with execution time $h(A)$, i.e $h(A)=\frac{\tau(T_b)}{\sigma_{k_b(A)}(T_b)}$. Let x be the number of *split* tasks in A.

Case 2.1: The algorithm terminated because $a=0$ in Step 2 or $a=1$ and $k_b(A)=1$ (i.e T_b is an *unsplit* job). The number of *split* tasks is p in the first case, and $p-1$ in the second case. Let z be the minimum execution time of a *split* task, and let T_l be a task realizing this minimum, i.e $z=\frac{\tau(T_l)}{\sigma_{k_l(A)}(T_l)}$.

It can be seen that $AVG(A)\ge z(1-1/p)+\frac{h(A)}{p}$. This is because there are at least $p-1$ *split* tasks with execution time not lesser than z, and a *split* or an *unsplit* task with execution

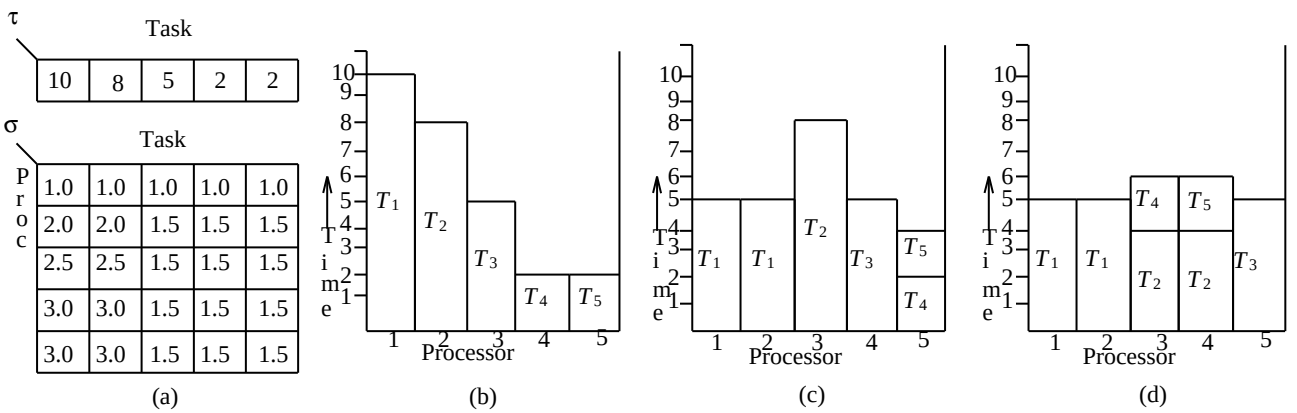

Fig. 1 Illustration of the part_schedule algorithm

time equal to $h(A)$. Since $f(A)=h(A)$, the above inequality is rewritten as $z \le \frac{(AVG(A)-\frac{f(A)}{p})}{(1-1/p)}$. Also, from Lemma 5, we have $2z \ge f(A)$. Combining the above two inequalities, we have $f(A) \le \frac{2(AVG(A)-\frac{f(A)}{p})}{(1-1/p)}$. This can be rewritten as $f(A) \le \frac{2AVG(A)}{(1+1/p)} \le \frac{2f(OPT)}{(1+1/p)}$, from Lemma 1 and 3.

Case 2.2: The algorithm terminated because $f(A) \ne h(A)$ in Step 3. However, this case has been considered in Case 1.

Case 2.3: The schedule C' constructed in Step 4 has $f(C') > h(A)$. The number of *split* tasks in C' is $x'=x+1$. Let y be the $p-x'+1$ largest execution time of an *unsplit* task in C'. Consider the optimum solution. From Lemma 2, $k_b(OPT) \ge k_b(A)$. If $k_b(OPT)=k_b(A)$, then from Lemma 1, $f(OPT) \ge \frac{\tau(T_b)}{\sigma_{k_b(OPT)}(T_b)} = h(A) = f(A)$. This implies $f(OPT)=f(A)$ contradicting our assumption. Thus $k_b(OPT) \ge k_b(A)+1 = k_b(C')$. Thus as in Lemma 3, we conclude $AVG(C') \le AVG(OPT) \le f(OPT)$. We have two subcases.

Case 2.3.1: The value y is less than or equal to the execution times of all the *split* tasks in C'. Similar to the proof of statement (2) of Lemma 6, we can prove that $f(C') \le \max\{h(C'), \frac{2AVG(C')}{(1+1/p)}\}$. From our assumption, $f(C') > h(A) \ge h(C')$. Thus $f(C') \le \frac{2AVG(C')}{(1+1/p)} \le \frac{2f(OPT)}{(1+1/p)}$. However, $f(A)=h(A)<f(C')$. The above two inequalities give the required result.

Case 2.3.2: The value y is greater than the execution times of some *split* tasks in C'. Let z be the minimum execution time of a *split* task in C', and let T_l be the task corresponding to it, i.e $z = \frac{\tau(T_l)}{\sigma_{k_l(C')}(T_l)}$. Clearly, in this case $y > z$. Thus z is less than or equal to the execution time of x' *split* tasks and $p-x'$ *unsplit* tasks. Also there is an extra task, corresponding to y. Thus $AVG(C') \ge z + y/p \ge z(1+1/p)$.

If $b \ne l$, T_l is assigned the same number of processors in schedules A and C'. Thus applying Lemma 5 to schedule A, we have $f(A) \le 2z$. In the case $b=l$, $f(A) = \frac{\tau(T_b)}{\sigma_{k_b(A)}(T_b)} \le \frac{2(T_b)}{\sigma_{k_b(A)+1}(T_b)} = 2z$, as $k_b(C')=k_b(A)+1$. Thus in both cases we have $f(A) \le 2z$. The above inequalities imply $f(A) \le \frac{2AVG(C')}{(1+1/p)} \le \frac{2f(OPT)}{(1+1/p)}$, giving the required result. Q.E.D.

The upper bound given by the above theorem is shown to be tight by the following example. Let $n=t$ and $p=2t-1$. Let $\sigma_j(T_i)=j$, for all i,j, where $1 \le i \le n$, $1 \le j \le p$. The value of $f(A)$ can be seen to be equal to 1. However, the optimal schedule is obtained by assigning $2t-1$ processors to each task. Thus $f(OPT)=\frac{t}{2t-1}$. Thus, $\frac{f(A)}{f(OPT)} = \frac{2t-1}{t}$. This can be rearranged as $\frac{2}{(1+1/(2t-1))}$.

Theorem 2: The *part_schedule* algorithm can be implemented in $O(n \log n + np \log p)$ time.

Proof: The proof is left to the reader. Q.E.D.

From the proof of Theorem 1, it is possible to design a much faster *approximate* algorithm for the *partitionable independent task scheduling* problem, that has the same worst case performance as the *part_schedule* algorithm. This algorithm, however, may give schedules which are worse when compared to the *part_schedule* algorithm.

procedure *fast_part_schedule*

[1] Set s_i to 1, for all $1 \le i \le n$, and set a to p ($p-a$ gives the current number of *split* tasks). Set *avg* to be $\frac{1}{p}\sum_{j=1}^{n}\tau(T_j)$, and *est* to be $\max\{h, avg + y(1-1/p)\}$, where h, y are the highest and $(p+1)^{th}$ largest execution time of a task respectively. Let T_i be a task with the highest execution time.

[2] If $a=0$ go to 5, else continue.

[3] If $a=1$ and $s_i=1$ then go to 5, else do the following. If $est \ne h$ then go to 5, else do the following.

[4] If $s_i=1$ set a to $a-2$, else set a to $a-1$. Increment s_i. Now let $T_{i'}$ be a task such that for all j, $\frac{\tau(T_{i'})}{\sigma_{s_i}(T_{i'})} \ge \frac{\tau(T_j)}{\sigma_{s_i}(T_j)}$, and let $h' = \frac{\tau(T_{i'})}{\sigma_{s_i}(T_{i'})}$. Also, let y' be the $(a+1)^{th}$ largest execution time of an *unsplit* task. Set *avg'* to $avg + \frac{1}{p}\left(\frac{\tau(T_i)s_i}{\sigma_{s_i}(T_i)} - \frac{\tau(T_i)(s_i-1)}{\sigma_{s_i-1}(T_i)}\right)$ and *est'* to $\max\{h', avg' + y'(1-1/p)\}$. If $h < est'$ then decrement s_i, restore a (i.e if $s_i=1$, set a to $a+2$, else set a to $a+1$), and go to 5. Set *est* to *est'*, *avg* to *avg'*, h to h' and i to i', and go to 2.

[5] Assign the jobs with $s_i > 1$ to $p-a$ processors. Assign the a largest *unsplit* tasks to the remaining a processors. The other *unsplit* tasks are assigned thereafter using a *list_schedule* algorithm.

end *fast_part_schedule.*

Example 2: Considering the problem instance of Fig. 1, the variables *avg*, h, i, y, *est*, are respectively set to 27/5, 10, 1, 0, and 10 respectively. As $est=h$, the check in Step 3 succeeds. Thus one more processor is temporarily allocated to T_1. The variables *avg'*, h', i', y', *est'* are respectively set to 27/5, 8, 2, 2, and 8 respectively. Since $est' < h$, T_1 is currently allocated 2 processors. The variables *avg*, h, i, *est* are set 27/5, 8, 2, and 8 respectively. As $est=h$, the check in Step 3 succeeds. Thus one more processor is temporarily allocated to T_2. The variables *avg'*, h', i', y', *est'*, are respectively set to 27/5, 5, 3, 2, and 7. Since $est' < h$, T_2 is currently allocated 2 processors. The variables *avg*, h, i, *est* are set to 27/5, 5, 3, and 7 respectively. Now the check in Step 3 fails as $est \ne h$. Thus a schedule is constructed in Step 5. This schedule is exactly the same schedule as in Fig. 1(d).

Theorem 3. For an instance, let A be the schedule output by the *fast_part_schedule* algorithm, and let *OPT* be an optimal schedule. Then, $f(A) \le \frac{2f(OPT)}{(1+1/p)}$.

Proof: The proof is similar to Theorem 1. Q.E.D.

The worst case example for the *fast_part_schedule* algorithm is the same as the one for the *part_schedule* algorithm.

Theorem 4. The *fast_part_schedule* algorithm can be implemented in time $O((n+p)\log p)$ time.
Proof: The proof is left to the reader. Q.E.D.

4. CONCLUSIONS

In this paper we have presented *approximate* algorithms for the *partitionable independent task scheduling* problem. We prove worst case bounds for each of the algorithms.

REFERENCES

[1] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, ''Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey,'' *Annals of Discrete Mathematics*, vol. 5, pp. 287-326, 1979.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. New York, N. Y.: W. H. Freeman and Company, 1979.

[3] K. P. Belkhale and P. Banerjee, ''A Parallel Algorithm for Hierarchical Circuit Extraction,'' *Int. Conf. on Computer Aided Design*, pp. 236-239, Nov. 1990.

[4] R. L. Graham, ''Bounds on Multiprocessing Timing Anomalies,'' *SIAM J. Appl. Math.*, vol. 17, pp. 263-269, 1969.