In Proceedings of the 1991 International Parallel Processing Symposium

A SCHEDULING ALGORITHM FOR PARALLELIZABLE DEPENDENT TASKS

Krishna P. Belkhale Prithviraj Banerjee

1101 W. Springfield Av.
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 task graph on a multiprocessor, called the *parallelizable dependent task scheduling* problem. Associated with each task, we are given the time it takes to run on a uniprocessor, and the speedup that can be obtained by running it on i processors, $1 \le i \le p$. We present an algorithm for the problem and analyze the performance.

1.INTRODUCTION

The problem that we are interested in will be called the *parallelizable dependent task scheduling (PDTS)* problem. It involves scheduling a directed task graph containing n tasks, $T_1, \cdots T_n$, on p processors. The directed arcs of the task graph represent the precedence relations between the tasks. An arc from T_i to T_j indicates that T_j can start only after the completion of T_i. The time that each task T_i takes to run on a uniprocessor is assumed to be known *a priori*, and is denoted by τ_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 $\sigma_i(k)$ 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 $\frac{\tau_i}{\sigma_i(k)}$ when run on k processors. An example task graph, with the associated time and speedup functions, is shown in Fig. 1. Referring to the figure, task T_4 can start only after T_1 is completed, T_5 can start only after T_1 and T_3 are completed, and T_6 can start only after T_2 and T_3 are completed.

A schedule S is defined as a specification of tasks to processors. For each task, it specifies the starting time for the task and the processors that it should be run on. This specification should satisfy the precedence constraints of the task graph. Also, at any time instant a processor can be assigned to at most one task. 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 $f(S)$.

This problem arises in many applications. For example, parallelizing compilers take an input program and construct the corresponding task graph [3]. Each of the task nodes could correspond to loops that could be parallelized across processors. The parallelizing compiler has to determine the number of processors to be allocated to each of the task nodes, and the schedule to minimize the total execution time.

This problem also arises in parallel computer-aided design applications [4]. Based on the structure of the input circuit, hierarchical partitioning methods can be used to subdivide the given task, into a set of subtasks with dependencies. Example applications include Compaction, Placement and Routing, Extraction and Design Rule Checking. The subtasks could have varying computational requirements, which can be usually estimated. Each of the subtasks could be run on more than one processor, using an appropriate parallel algorithm. In order to combine the benefits of hierarchy and parallelism, we need to assign processors to the subtasks so that the total execution time is minimized.

In [5] we have described an approximate algorithm for the problem, for the special case when the tasks are all independent (*PITS*). In Section 2, we briefly review

* This research was supported in part by the National Science Foundation Award under Grant NSF MIP 87-14646, and in part by NASA under Grant NAG 1-613.

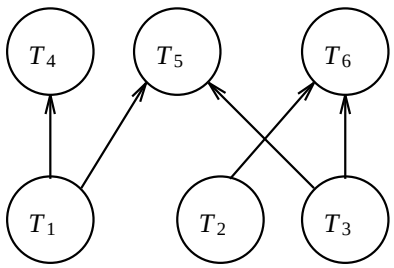
the approximate algorithm for the special case. In Section 3, we describe an algorithm for the general case of the problem, when the tasks are dependent. In Section 4, we analyze the performance of the algorithm.

2. REVIEW OF PITS

The *parallelizable independent task scheduling (PITS)* problem allows the possibility of executing a task among multiple processors. The problem is NP-hard as the simpler problem of scheduling non parallelizable independent tasks is NP-hard [2].

In this section, we review the *approximate* algorithm presented in [5]. The main idea in the algorithm is to construct an *LPT* schedule [6] (tasks are assigned to idle processors in decreasing order of task computation time) 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 *LPT* schedule time. This is a conservative strategy. The intuitive reason behind this is that increasing the number of processors allocated to a task will lead to an increase in the cumulative load of all the tasks, which could potentially increase the schedule time. The complete algorithm is described below.

$n = 6, p = 5$



(a)

τ	Task					
	5	5	10	20	5	10

σ	Task					
1	1	1	1	1	1	1
proc 2	20/11	20/11	20/11	20/11	20/11	20/11
3	5/2	5/2	5/2	5/2	5/2	5/2
4	3	3	40/13	40/13	3	40/13
5	3	3	25/7	25/7	3	25/7

(b)

Fig. 1(a). Task graph (b) Time and speedup

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 $a = p - \sum_{s_i > 1} s_i$). 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, $1 \le j \le n$, $\frac{\tau_i}{\sigma_i(s_i)} \ge \frac{\tau_j}{\sigma_j(s_j)}$. Let $h = \frac{\tau_i}{\sigma_i(s_i)}$.

[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.*

Some intuitive explanations follow. In Step 3, the conditional statement $f(C) \ne h$ checks if the task T_i is a bottleneck with the current number of allocated processors. If it is, Step 4 checks if allocating one more processor to T_i degrades the overall schedule time. If it does not, one more processor is allocated to T_i and the current schedule is updated.

Example 1: We illustrate the algorithm using the problem instance shown in Fig. 2(a). The initial *LPT* schedule time is 11 (processor 1 gets T_1, 2 gets T_2, 3 gets T_3, 4 gets T_4, and 5 gets T_5 and T_6). Going through the steps, T_1 is selected in Step 2, i.e $i = 1$, and h is set to 11. The check in Step 3 succeeds as the current solution is 11, which is equal to h. The task T_1 is temporarily assigned 2 processors, and the check in Step 4 is performed. It can be seen that all the tasks can be scheduled within time 9 (processor 1 gets T_1, 2 gets T_1,

3 gets T_2, 4 gets T_3 and T_5, and 5 gets T_4 and T_6), which is not higher than 11. 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 9. The check in Step 3 succeeds as the current solution is 9, 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. 2(b), all the tasks can be scheduled within time 7, which is not higher than 9. Thus T_2 is currently assigned 2 processors. Next, the task T_1 is selected with $h=6$. However, the current solution is 7, which is not equal to h. The algorithm is hence terminated, with the schedule in Fig. 2(d) as the solution.

Let $\eta_i(j)=\frac{\sigma_i(j)}{j}$. The condition that $\eta_i(j)\geq\eta_i(j+1)$, for all i, j, where $1\leq i\leq n$ and $1\leq j\leq p-1$, is called the *efficiency* condition. It is very likely that this condition holds, as it can be easily proved that if the speedup curve of a task has a concave shape then it satisfies the efficiency condition, though the converse is not true. For an instance of the problem, let A denote the the schedule given by the *part_schedule* algorithm, and let *OPT* denote an optimum schedule. The following theorem, proved in [5], bounds the performance of the *part_schedule* algorithm.

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, $f(A)\leq\frac{2f(OPT)}{(1+1/p)}$.

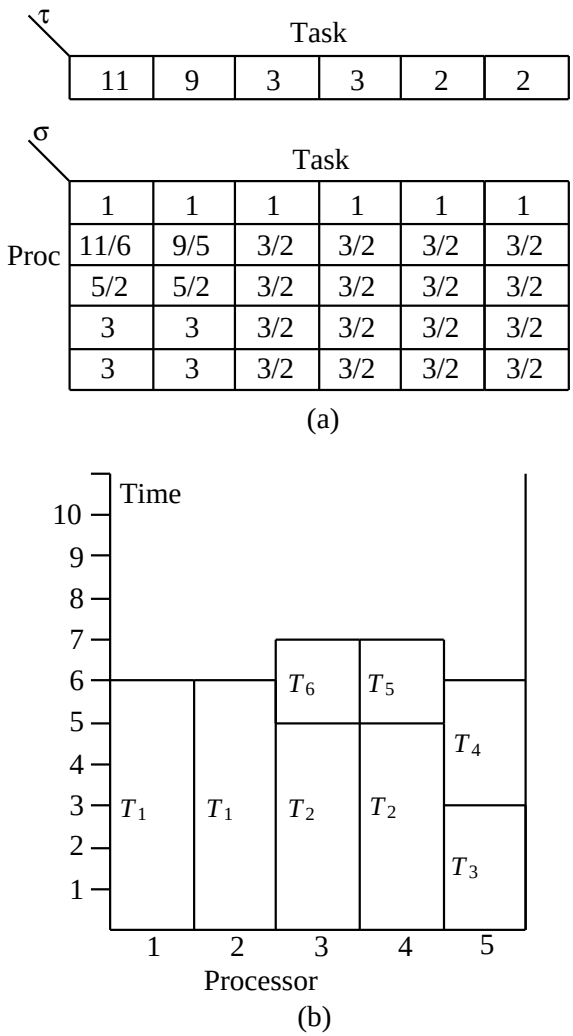


Fig. 2. Illustration of the part_schedule algorithm

3. ALGORITHM FOR PDTS

One approach to solve the parallelizable dependent task scheduling problem is to use an algorithm called the critical path algorithm [7]. In this algorithm, for each task node, we determine the length of the longest path (i.e add the times of the task nodes in the path) in the task graph starting from the task node. This is assigned as the priority value of the task. The schedule is then constructed by iteratively choosing a schedulable task (i.e its predecessor tasks have completed) with the highest priority and assigning it to a free processor.

The above approach has the limitation that the final schedule time is bounded below by the length of the longest path in the task graph, called the critical path. Thus it may be advantageous to assign more than one processor to a task, as this reduces the critical path time. The assigned processors can perform the task in parallel. However, since the speedups obtained will be less than linear, assigning many processors to a task could result in an increase in the overall time.

The *PDTS* problem is a NP-hard problem, as the simpler problem of scheduling tasks, where the tasks can run on only one processor and have no dependencies, is NP-hard. In this section, we give an algorithm for the problem. The algorithm has two steps. In the first step, called the allocation step, we determine the number of processors to be assigned to each of the tasks. The second step, called the mapping step, will be to actually construct the schedule from the output of the first stage. In the next two subsections, we describe the issues in both the steps.

3.1. The Allocation Step

One of the objectives in the allocation step is to minimize the critical path time of the task graph by

assigning more processors to tasks, which tends to reduce the final schedule time. However, assigning more processors to tasks increases the average load on a processor, as the speedups obtained are less than linear, and hence tends to increase the final schedule time. For example, in Fig. 1, the average load and critical path time are 11 and 25 respectively. If T_4 is assigned two processors, its execution time will reduce to 11. The critical path time will reduce to 16. However, since T_4 has been replaced by two tasks taking 11 units of time each, the average load on a processor is increased to 11.4. Thus we need to assign processors so that the maximum of the critical path and the average load on a processor is minimized. This problem can be formulated by the following integer non linear program.

Minimize ϕ, *subject to the following constraints*

(1) $\phi \geq y_i, 1 \leq i \leq n.$

(2) $\phi \geq \frac{1}{p}\left(\sum_{1 \leq i \leq n} \frac{s_i \tau_i}{\sigma_i(s_i)}\right)$

(3) $y_i - y_j \geq \frac{\tau_i}{\sigma_i(s_i)}$, *for all* i, j, *where* T_i *depends on* T_j.

(4) $y_i \geq \frac{\tau_i}{\sigma_i(s_i)}$, *for all* i.

(5) $1 \leq s_i \leq p$ *and integral , for all* i.

Here, y_i gives the earliest finish time of task T_i, given an infinite number of processors. The variable s_i denotes the number of processors allocated to task T_i. The conditions in (1), (2) imply that we are minimizing the maximum of critical path time and the average. The conditions in (3) are due to precedence constraints between the tasks. It relates the earliest finish time of task T_i in terms of the earliest finish times of its predecessors.

The integrality condition of (5) could be omitted, leading to a non linear optimization problem. For this problem, we assume that the speedup functions $\sigma_i(x)$ are given as functions over reals. The solution to the non linear optimization problem can then be rounded to get a close answer to the integer non linear optimization problem described above. Thus we will concentrate on the non linear optimization problem in rest of this section.

Consider the case when the speedup functions are concave, which is usually the case. It can be easily proved that if $f(x)$ is concave and positive over a domain, then $1/f(x)$ is a convex function over the domain. Thus the functions $\frac{1}{\sigma_i(x)}$ are all convex. Thus constraints in (3) (4) are of the form $g(x) \leq 0$, where $g(x)$ is a convex function. The constraint (2) can also be cast in this form if the functions $\frac{x}{\sigma_i(x)}$ are all convex. This is a reasonable condition, as $\sigma(x)$ by Amdahl's law [8] is usually given as $\frac{1}{(1-\alpha)/x + \alpha}$, where α is the fraction of the program that is inherently serial. The function $\frac{x}{\sigma(x)} = (1-\alpha) + \alpha x$, which is convex. With this extra condition, the nonlinear optimization problem is essentially minimizing a convex function over a closed convex set. Hence, the optimum solution can be obtained by standard algorithms such as steepest descent algorithms [9]. If the two conditions described above are not valid, non linear optimization procedures can only guarantee a relative minimum.

3.2. Mapping Step

The second step is to construct the schedule from the processor allocation information provided by the allocation step. A fundamental issue in this step is the following. A task T_i can start only after all the tasks it depends on are completed. Also, the task T_i may be delayed further until the required s_i processors are available. Greater the value of s_i, greater will be the waiting time, which may result in an increase in the overall schedule time, especially if no other task can be scheduled during this waiting period. Thus it may be necessary to modify the processor allocation during the mapping step. Also, decreasing the value of s_i, while decreasing the waiting time, may increase the critical path time of the task graph, and hence may increase the overall schedule time.

We present a simple algorithm below for the mapping step, whose function depends on a parameter $0 < \gamma < 1$.

procedure *map*

[1] For all tasks T_i, set $a_i = s_i$, and if $a_i > \lceil \gamma p \rceil$ then reduce a_i to $\lceil \gamma p \rceil$.

[2] Build the schedule iteratively, by scheduling a new task T_i whenever all its predecessors have been completed, and a_i processors are available.

end *map*

The schedule constructed in step (2) has the property that, at any time instant, the number of idling processors is less than the number of processors required by any of the unfinished tasks that are ready to start (i.e its predecessor tasks have completed). In other words, there is no forced idleness. Also, the second step of the algorithm can be performed dynamically at run time, when the tasks are actually executed.

4. ANALYSIS OF THE ALGORITHM

The choice of γ is crucial for the performance of the *map* algorithm. The following theorem gives the value of γ that realizes the best worst case performance.

Theorem 2: Assuming that the efficiency condition is true, the value $\frac{(3-\sqrt{5})}{2} \approx .382$ is the best choice for γ. It ensures that the finish time $f(M)$ of the schedule M produced by the *map* algorithm is within $\left\lfloor \frac{p}{\frac{(\sqrt{5}-1)p}{2}} \right\rfloor A + C$, where A is the average and C is the critical path time of the task graph with the processor allocation as specified at the end of the allocation step.

Proof: Let us suppose that the cutoff used in step (1) of the algorithm is x. Consider the average load A' on a processor in schedule M, i.e $A' = (1/p) \sum_i \frac{a_i \tau_i}{\sigma_i(a_i)}$. Clearly, $s_i >= a_i$. Hence, from the efficiency condition, $A' \le (1/p) \sum_i \frac{s_i \tau_i}{\sigma_i(s_i)} = A$.

For each task T_i, we define a value t_i based on the output of the allocation phase. This represents the earliest start time of T_i. Thus, the value of t_i is 0 for tasks which do not depend on other tasks, and $\max_j \{ t_j + \frac{\tau_j}{\sigma_j(s_j)} \mid$ there is an arc from T_j to $T_i \}$ otherwise.

Consider the schedule M. At any time t, we define a quantity $v(t)$ called the virtual time. Let U be the set of tasks that have not been started till time t, and let E be the set of tasks that are currently executing at time t. For each task T_i in E, let α_i denote the fraction of the task T_i that has been completed at time t. Then, $v(t)$ is defined as $\min\{ \min_{T_i \in U} t_i, \min_{T_i \in E} (t_i + \alpha_i \frac{\tau_i}{\sigma_i(s_i)}) \}$. Clearly, the value of $v(t)$ at the end of the schedule is C.

It should be noted that the virtual time function $v(t)$ is defined in terms of the task graph at the end of the allocation phase. The value $v(t)$ is a measure of the portion of the task graph that has been completed at time t. It can be easily verified that if $v(t) \ge t_i$, then by time t all the predecessor tasks of task T_i have been completed in M.

We now use an amortization argument to prove the result of the theorem. Let $\mu = \frac{p}{p-x}$, and let $T = \mu A + C$. We want to find the smallest μ so that $f(M)$ is upper bounded by T. Consider any interval of time from t to $t + \Delta t$, during which the allocation of tasks to processors in M does not change. There are two cases.

Case 1: The virtual time does not increase in the interval. This implies the existence of a task T_i, with $v(t) = t_i$, that has not been executed till $t + \Delta t$. Since $v(t) = t_i$, all predecessor tasks of T_i have finished execution by time t. Thus T_i is ready for execution at time t. Since T_i requires at most x processors, it implies that at least $p - x$ (actually $p - x + 1$) processors are busy during the interval. This is illustrated in Fig. 3(a). Thus the average load of all the processors during the interval is at least $\Delta t \frac{p-x}{p}$. Thus the time Δt can be charged to $\frac{p}{p-x} A' = \mu A'$. However, since $A' \le A$, Δt can be charged to μA.

Case 2: The virtual time increases in the interval. Without any loss of generality, we can assume that the increase is due to the execution of a single task T_i. This is illustrated in Fig. 3(b). The fraction of T_i that is executed in the interval is $\Delta\alpha_i = \frac{\Delta t}{\tau_i / \sigma_i(a_i)}$. Thus the increase in virtual time in the interval is $\Delta\alpha_i \frac{\tau_i}{\sigma_i(s_i)} = \Delta t \frac{\sigma_i(a_i)}{\sigma_i(s_i)}$. Since $a_i \le s_i$, applying the efficiency condition, we observe that the increase in virtual time is not less than $\Delta t \frac{a_i}{s_i}$.

If $s_i \le x$, then $a_i = s_i$. Thus the increase in virtual time is the same as Δt. Thus Δt can be charged to the increase in virtual time, and hence to the critical path C.

In the other case $s_i > x$, implying $a_i = x$. The increase in virtual time is at least $\Delta t \frac{x}{s_i}$. Thus Δt cannot be fully charged to the increase in virtual time. There is also a component due to the μA product. The contribution of task T_i to the average A, in the interval t to $t + \Delta t$, is $\Delta\alpha_i \frac{s_i \tau_i}{p \sigma_i(s_i)}$. Thus the virtual time increase, plus μ times the average, together provide a contribution that is at least $\Delta t (\frac{x}{s_i} + \frac{x\mu}{p})$. This has to be at least Δt. In other words, $\Delta t \le \Delta t (\frac{x}{s_i} + \frac{x\mu}{p})$. This can be guaranteed if $1 \le \frac{x}{p} + \frac{x\mu}{p}$. Substituting $\mu = \frac{p}{p-x}$, we solve for the smallest x (implying smallest μ) that satisfies the above condition. Simplifying, we get the inequality $x^2 - 3px + p^2 \le 0$. The smallest integer x satisfying the above equation is $\left\lfloor p \frac{(3-\sqrt{5})}{2} \right\rfloor$. This proves the theorem. □

Thus, asymptotically the schedule time of the *map* algorithm is within $\frac{2}{\sqrt{5}-1} A + C$, that is within $1.62A + C$. It is very likely that we can design more sophisticated theoretical and practical algorithms for the mapping problem, which adjusts the number of processors allocated to a task at the time the schedule is being

constructed, instead of having a maximum cut off. We are currently investigating algorithms of this nature. However, these algorithms may only allow for a static construction of the schedule. The *map* algorithm described above represents a simple algorithm that shows the different tradeoffs involved in the mapping problem.

Finally, we want to analyze the complete algorithm for the *PDTS* scheduling problem. The time taken by an optimal schedule is at least $\max(A^{opt}, C^{opt})$, where A^{opt}, C^{opt} are the average and the critical path time corresponding to the optimal schedule. However, if the allocation step is carried out optimally $\max(A, C) \leq \max(A^{opt}, C^{opt})$, where A, C are the average and the critical path time of the task graph, with processor allocation as specified at the end of the allocation step. Thus combining these observations, we have the following theorem.

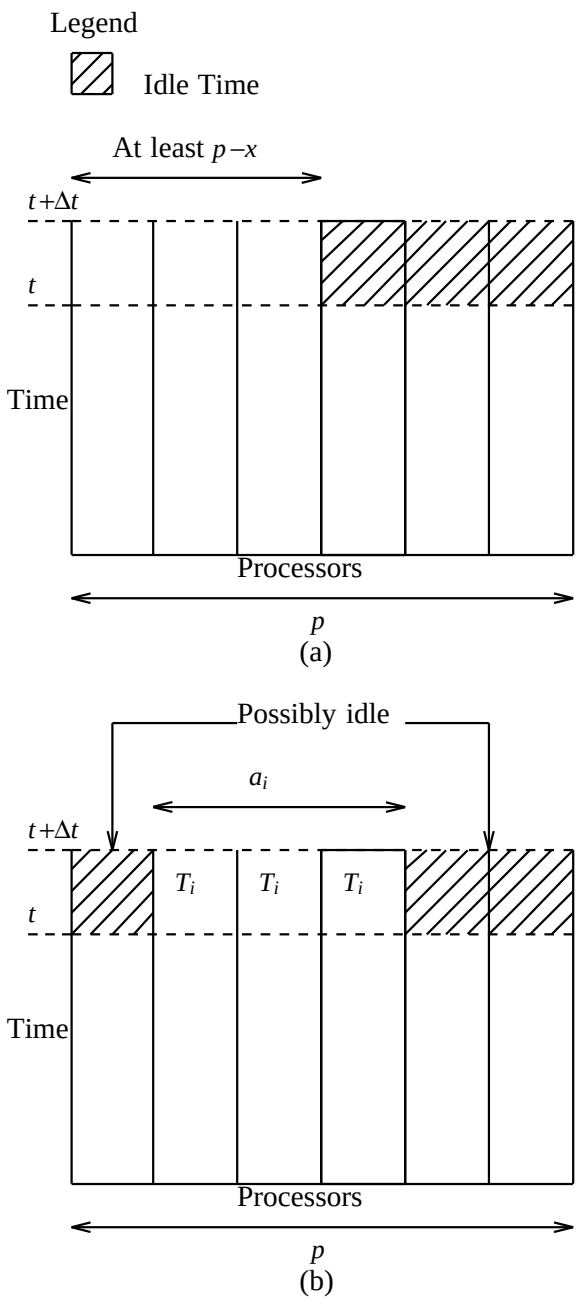


Fig. 3(a) Case 1 (b) Case 2 of Theorem 1

Theorem 3: If the allocation step is performed optimally and the efficiency condition is true, the schedule time of the complete algorithm is within $\left\lceil \frac{(\sqrt{5}+1)p}{2} \right\rceil \Big/ \left\lceil \frac{(\sqrt{5}-1)p}{2} \right\rceil \approx 2.62$ times the optimal schedule time.

Example 2: Consider the problem instance in Fig. 1. There are 6 tasks, and the task graph needs to be scheduled onto 5 processors. Initially, the average load and critical path time of the task graph are 11 and 25 respectively. In Fig. 4(a), we show a possible result of the allocation phase. The number of processors assigned to a task, and the corresponding time for the task, are shown in the figure. The average load and critical path time are respectively 12.3 and 10.75 respectively. In Fig. 4(b), we show the final schedule constructed by the *map* algorithm using a cutoff of $\lceil 0.382 * 5 \rceil = 2$. The final schedule length is 13.75. For comparison, if a task cannot be assigned to multiple processors, the optimal schedule time is 25.

5. CONCLUSIONS

In this paper we have introduced a new problem in scheduling called the parallelizable dependent task scheduling problem. This problem arises in different situations, such as in parallelizing compilers and in parallel computer-aided design. We have described an algorithm for the problem, and analyzed its performance. Future research on this problem should concentrate on improving the performance of the algorithms, both from theoretical and practical perspectives.

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.

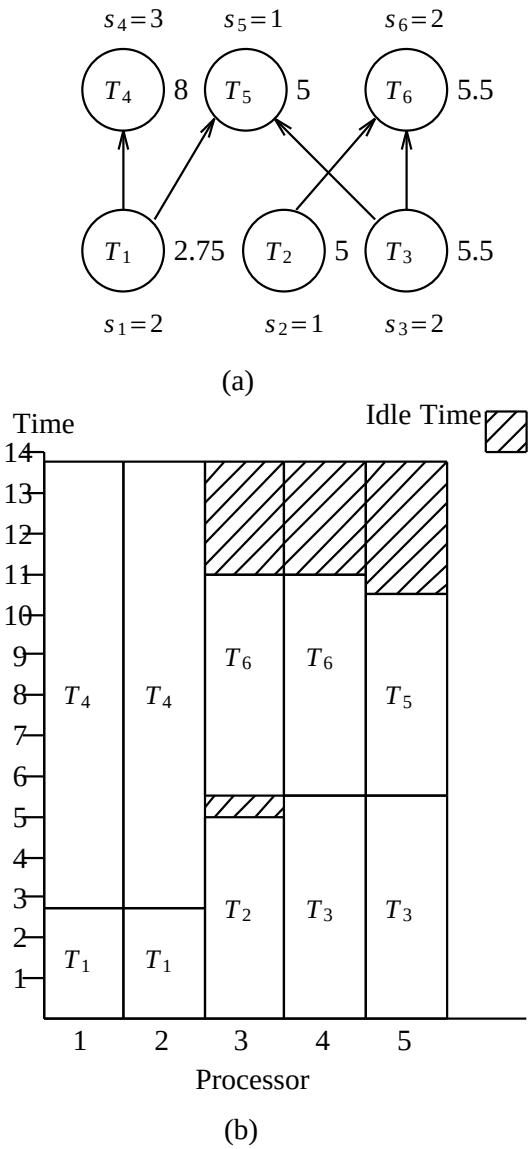


Fig. 4 (a) Allocation step (b) Mapping Step.

[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] C.D. Polychronopoulos and U. Banerjee, ‘‘Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds,’’ *IEEE Trans. on Computers*, vol. c-36(4), pp. 410-420, April 1987.

[4] Krishna P. Belkhale and Prithviraj Banerjee, ‘‘Tradeoffs in Parallelism and Hierarchy in speeding up VLSI CAD applications,’’ *IEEE Trans. Computer Aided Design*, vol. submitted.

[5] K. P. Belkhale and P. Banerjee, ‘‘Approximate Algorithms for the Partitionable Independent Task Scheduling Problem,’’ *Proc. Int. Conf. on Parallel Processing*, pp. 72-75, Aug. 1990.

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

[7] N. F. Chen and C. L. Liu, ‘‘On a class of scheduling algorithms for multiprocessor computing systems,’’ in *Parallel Processing, Lecture Notes in Computer Science*, vol. 24, Berlin, pp. 1-16, 1975.

[8] G. M. Amdahl, ‘‘Validity of the single processor approach to achieving large-scale computing capabilities,’’ in *AFIPS Conference Proceedings*, vol. 30, Montvale, N.J, p. 483, 1967.

[9] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1973.