Task Scheduling Algorithm to Reduce the Number of Processors using Merge Conditions

Tae-Young Choe
Dept. of Computer Engineering
Kumoh National Institute of Technology
Gumi, South Korea
choety@kumoh.ac.kr

***Abstract*—Some task scheduling algorithms generate the shortest schedule, when its input DAG satisfies a specified condition. Among those scheduling algorithms, TDS algorithm proposed a DAG condition where allocation of two parent tasks of a join task in the same processor cause longer schedule length than allocation in different processors, and it generates the shortest schedule if any input DAG satisfies the condition. In the paper, we propose a post-processing scheduling algorithm that reduces the number of processors while preserving its schedule length. Especially, we propose conditions where two processes can be merged without increasing schedule length. Experimental results show that the number of processor is reduced to 92.3% ~ 98.0% if schedule length is reserved and required computing power is reduced to 84.3% ~ 91.2% if schedule length can be increased.**

Keywords-task scheduling; processor reduction; merge condition; DAG; TDS algorithm; post-processing

I. INTRODUCTION

Recently, trend of computer architecture is moving from high performance single core CPU system to multicore CPU systems. This tendency includes PC having Dual-core or Quad-core CPU and server computers moving from mainframe to Grid Computing or Cloud Computing. However, since performance of parallel processor is highly affected by methods of dividing application and allocating them to processors, the assignment methods are important components in multi-processor systems.

Until now, various parallel algorithms are developed and ported to parallel computer systems. Unfortunately, the parallel algorithms should be designed directly by programmer, which makes it hard for general programmer to manipulate various parallel system structures. In order to divide an application and to allocate them to multiple processors, compiler and loader have been developed. They divide the applications to tasks, define messages between tasks, and allocate the tasks to processors. Especially, task scheduling algorithms decide processors where tasks are allocated. Thus they take a major part that determines execution time of given application.

Task scheduling algorithm is known as an NP-hard problem [1]. However, because of its importance, lots of heuristic task scheduling algorithms are proposed [2, 3, 4, 5]. Among them, Darbha et al. proposed a task scheduling algorithm that generates the shortest schedule when a given directed acyclic graph (DAG) satisfies a specific condition [6] when task duplication is allowed. The task duplication is a policy where any task can be duplicated to multiple processors. The specific condition is a property of DAG where allocating two parent tasks of a join task in the same processor do not reduces schedule length.

In general, task duplication reduces schedule length because it eliminates some communication overheads. Although task duplication affects reduction of schedule length, duplicated tasks which exist in multiple processors erode available CPU times [7]. Thus it reduces efficiency of multi-processor system which runs multiple applications. TDS algorithm, typical task duplication scheduling algorithm, assumes that there are sufficient number of available processors. Thus schedules generated by such scheduling algorithm require lots of processors which are not suitable to real parallel systems.

Some task scheduling algorithms have tried to reduce the number of required processors [8, 9]. Doruk et al. proposed a task scheduling algorithm that reduces the number of processors using genetic algorithm [9]. Since the time complexity of the algorithm is $O(N^3)$ when the number of task is N, it is not suitable to huge size DAG. Shan and Choe tried to reduce the schedule length using a post processing algorithm that merges two parent processors of a join task in heterogeneous computing systems [8]. The post processing algorithm assumes heterogeneous computing system where it is very hard to obtain optimal schedule. So it tries to merge two processors where a communication exists between them without maintaining schedule length.

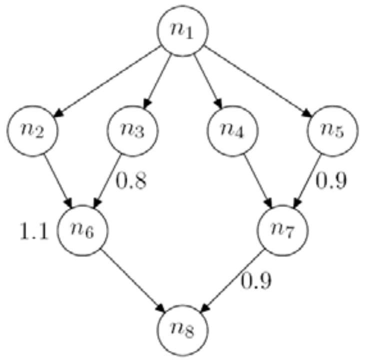


Figure 1. An example of DAG. Weight of node or edge where the value is not specified is assumed to be 1.

However, processors which have no communication between them can be merged, which reduces the number of required processors. In this paper, we propose a condition that checks properness of merge between two processors. And we propose an algorithm that finds merge pairs based on the condition. After it finds a set of merge pairs, it merges them. The algorithm includes a function that finds maximum pair in order to reduce the number of merge.

The rest of the paper is organized as follows. Section 2 defines terms and objectives. Section 3 introduces related previous works and discusses their problems. Section 4 presents conditions where merge operation does not increase schedule length. Section 5 explains details of the proposed algorithm. Section 6 shows experimental results of the algorithm. Finally, the paper concludes in Section 6.

II. SYSTEM MODEL

We define a *task* as a continuous list of codes that should be executed sequentially. In the case, a program is composed of multiple tasks and data transfers between tasks if the task needs results from its previously executed tasks. If we assume a task as a *node* and a data transfer as a directed *edge*, we can express an application as a directed graph, which is composed of a set of nodes n_i and a set of directed edge $e_{i,j}$ which starts from node n_i and arrives to node n_j.

Since an edge in a DAG is directed, a node has two types of adjacent edges: one is an incoming edge and the other is an outgoing edge. *Incoming degree* of a node is the number of incoming edges and *outgoing degree* of a node is the number of outgoing edges of the node. *Degree* of a node is the sum of above two degrees. *Join task* is a task of which incoming degree is larger than 1 and *fork task* is a task of which outgoing degree is larger than 1. *Entry task* is a task of which incoming degree is zero and *exit task* is a task of which outgoing degree is zero. In order to simplify the problem, we assume that DAG has a single entry task and a single exit task. For example, if there are multiple exit tasks, all the exit tasks are connected to a dummy task, which becomes a new single exit task. Figure 1 shows an example of DAG that one entry task and one exit task, where node n_1 is the entry task and node n_8 is the exit task. Weight of a node represents the execution time of the task and weight of an edge means amount of information from a parent task to its child task. Execution time of task n_i is notated as τ_i and communication time of edge $e_{i,j}$ is notated as $c_{i,j}$. In Fig. 1, weight of all nodes except n_6 is 1 and weight of all edges is 0.9 if its weight is not specified. Weight of task n_6 is 1.1. Weights of edges $e_{3,6}$, $e_{5,7}$, and $e_{7,8}$ are 0.8, 0.9, and 0.9, respectively.

In order to run an application in multi-processor system, tasks of the application should be allocated to available processors. After an allocation, each processor contains a list of tasks, because the tasks should be executed in an order. In the paper, a processor is expressed as $C(n_i)$ if the last task in the processor is n_i. *Task scheduling* is to allocate tasks of an application to processors in order to minimize completion time. A task in a processor has a start time and a completion time. For a task n_i in a processor C_i, start time and completion time are notated as $st_{Ci}(n_i)$ and $ct_{Ci}(n_i)$, respectively. In general, static task scheduling does not consider preemption in order to eliminate overhead by preemption. Thus completion time of a task is sum of its start time and weight of the task. That is,

$$ct_{c_i}(n_i) = st_{c_i}(n_i) + \tau_i .$$

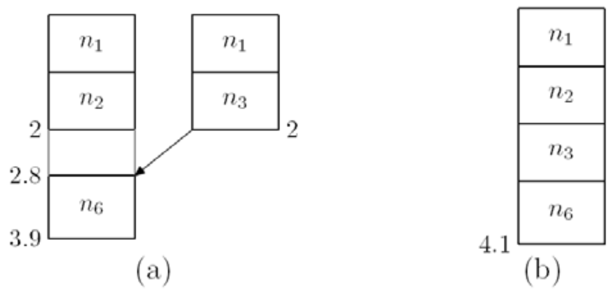


Figure 2. Four types of task allocation in a join task : (a) parent tasks of join task n_6 are allocated

Before a task starts, its entire parent tasks should finish and their results should arrive. The time that result of a parent task arrives is sum of a completion time of the parent task and a transfer time of the result. The sum is called *ready time*. If a task and a parent task are allocated in the same processor, the transfer time is assumed to be zero because the communication can be implemented as very fast operations like memory copy. For a task n_i in processor C_i, if its child task n_j is in processor C_j and edge $e_{i,j}$ connects from n_i to n_j, ready time $rdy_{C_iC_j}(n_i,n_j)$ is defined as follows:

$$rdy_{C_i,C_j}(n_i,n_j) = \begin{cases} ct_{C_i}(n_i), & \text{if } C_i = C_j \\ ct_{C_i}(n_i) + c_{i,j}, & \text{otherwise} \end{cases}.$$

If a task in processor C_i needs to wait a message from other processor, there could be an interval in C_i where no task runs. The interval is called *empty slot*. In Fig. 2 (a), there is an empty slot of size 0.8 between task n_2 and n_6. If task duplication is allowed, there could be multiple messages from duplicated parent tasks. Among them, the parent task that has the earliest finish time sends the message to the child task. That is, what we want is the minimum ready time among duplications of the parent task. Thus, for task n_a in processor C_a and its parent task n_i, ready time $rdy_{Ca}(n_i,n_a)$ is computed as follows:

$$rdy_{C_a}(n_i,n_a) = \min_{C_a} \min_{C_i \ni n_i} rdy_{C_i,C_a}(n_i,n_a).$$

Task n_a starts after all the ready time of all its parent tasks. For a task n_a in processor C_a, start time $st_{Ca}(n_a)$ is defined as follows:

$$st_{C_a}(n_a) = \max_{n_i \in pred(n_a)} rdy_{C_a}(n_i,n_a),$$

where $pred(n_a)$ is a set of parent tasks of n_a.

After setting the start time of the entry task as 0, the completion time of the exit task is called *schedule length* or *makespan*. Schedule that has the shortest schedule length given a DAG is called *optimal schedule*. Algorithm that finds a task schedule is called *task scheduling algorithm* and the objective of the algorithm is to generate the shortest schedule given DAG.

If a task can be duplicated in multiple processors, schedule length can be reduced. For example, a schedule of DAG in Fig. 1 with task duplication is shown in Fig. 3 (a). In the Figure, by being duplicated task n_1 to $C(n_3)$ and $C(n_7)$, tasks n_3 and n_4 start earlier. Otherwise, start time of n_3 would be 2 not 1.

III. PREVIOUS WORKS

Darbha et al. proposed a condition which is required for optimal schedule and task scheduling algorithm that generates an optimal schedule given the condition. The algorithm is called Task Duplication based Scheduling (TDS) algorithm and it assumes that sufficient number of processors are provided and task duplication is allowed [6]. The condition means that given any join task, its parent tasks should be allocated to different processors in order to make the shortest schedule. For example, consider a sub-DAG consist of nodes n_1, n_2, n_3, and n_4 in Fig. 1. By comparing a task allocation to two processors as in Fig. 2 (a) and a task allocation to a single processor as in Fig. 2 (b), we know that the first case has shorter schedule length. If weight of $e_{3,6}$ is 1.1 instead of 0.8, the completion time of n_6 becomes 4.2. On the other hand, after all tasks being allocated to the same processor as shown in Fig. 2 (b), completion time of n_6 becomes 4.1, which is shorter than the previous allocation. Notice that transposition between n_2 and n_3 in Fig. 2 (b) does not modify schedule length.

After all, if parent tasks of a join task are allocated to different processors, start time of the join task is fixed as the minimum value. Since the condition assumes that all processors has the same computing power, the start time of a task n_i is the same and it is notated as *est*(n_i). Likewise, the completion time is notated as *ect*(n_i). Following terminologies given join task n_a use the same notations in [6]:

pred(n_a) : a set of parent tasks of task n_a,

rdy(n_i,n_a) : (ready time) completion time of message transfer from n_i to n_a,

fpred(n_a) : the task that has the largest ready time among parent tasks of n_a,

spred(n_a) : the task that has the second largest ready time among parent tasks of n_a.

For example, *pred*(n_6) = {n_2, n_3}, *rdy*(n_3,n_6) = 2.8, *pred*(n_6) = {n_2, n_3}, *rdy*(n_3,n_6) = 2.8, *fpred*(n_6) = n_2, and *spred*(n_6) = n_3 are shown in Fig. 1. Start time *est*(n_a) and completion time *ect*(n_a) of task n_a is defined as follows:

$$ect(n_a) = est(n_a) + \tau_a ,$$
$$est(n_a) = \max(\ ect(fpred(n_a)), rdy(spred(n_a), n_a))$$

where *rdy*(n_i,n_a) = *ect*(n_i)+$c_{i,a}$. TDS algorithm generates an optimal schedule if given DAG satisfies the following Darbha's Condition.

Condition 1 (Darbha's Condition). *Given join task n_a, let n_1 and n_2 be parent tasks of n_a, where n_1 = fpred(n_a) and n_2 = spred(n_a).*

$$\begin{array}{ll} \text{if } est(n_1) \ge est(n_2), & \tau_1 \ge c_{2,a}, \\ \text{otherwise} & \tau_1 \ge (c_{2,a} + est(n_2) - est(n_1)) \end{array}$$

TDS algorithm allocates parent tasks of the join task to different processors and let the task be executed at its *est*() time.

Park and Choe proposed a task scheduling algorithm that generates shorter schedule by merging than TDS algorithm does if given DAG satisfies more restricted condition [10]. The condition means that completion time of the last task in the merged processor is smaller than sum of successor task weights and communication time, after merging all tasks in a processor to another processor. The condition has little relation with Darbha's Condition and the range of the value of the condition is narrower than that of Darbha's Condition.

Shen and Choe proposed HPSA algorithm that merges scheduled processors in order to map a schedule to restricted number of processors [8]. HPSA algorithm considers heterogeneous computing system. Instead of generating optimal schedule, the algorithm tries to reduce schedule length where the restricted number of processors is provided. Since the algorithm focuses on pairs of processors connected by edges, the number of considered cases is relatively small.

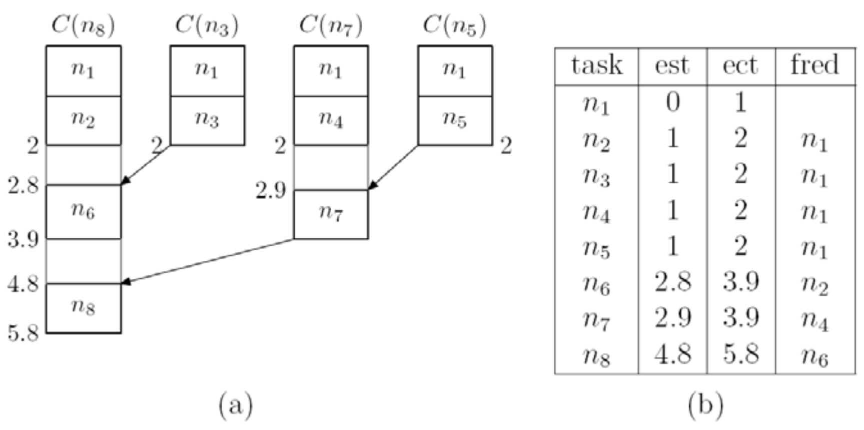
Bozdağ et al. proposed Schedule Compaction (SC) algorithm that merges scheduled processors in order to reduce the number of processor to the number of required size [9]. SC algorithm composes of following 3 phases: in the first phase, the algorithm computes characteristics of each task; in the second phase, it eliminates unnecessarily duplicated tasks based on the characteristics computed in the previous phase; in the third phase, merge between two processors are repeated. The repetition stops if there is no processor pair (C_i, C_j) which satisfies following merge condition:

$$\sum_{n_i \in C_i} \tau_i + \sum_{n_j \in C_j} \tau_j - \sum_{n_k \in C_i \cap C_j} \tau_k \le \max(\ ct(C_i), ct(C_j))$$

where *ct*(C_i) means completion time of the last task in processor C_i. SC algorithm decides a pair of processors as mergeable if execution time of tasks in the merged processor is earlier than the unmerged case. SC algorithm gives higher priority to processor pairs which share more tasks. Unfortunately, SC algorithm does not consider empty slot created by delayed tasks, which can increase completion time in merged processor. On the other hands, the idea that considers all possible processor pairs gives good result of merging in task scheduling. We apply the idea more rigorously in order to reduce possibility of schedule length increase.

IV. CONDITIONS FOR REDUCING THE NUMBER OF PROCESSORS

TDS algorithm generates the shortest schedule by allocating parent tasks to different processors while overusing processors. In the paper, we propose a method that reduces the number of required processors while maintaining its schedule length. Fig. 3 (b) shows tasks information of DAG in Fig. 1 used by TDS algorithm. Ready time from task n_2 to join task n_6, *rdy*(n_2,n_6), is 3 and *rdy*(n_3,n_6) is 2.9. Join task n_6 is allocated to processor *C*(n_6) where n_2 is assigned because n_2 has the largest ready time. On the other side, task n_3 is allocated to other processor *C*(n_3) because it has the second largest ready time and satisfies Darbha's Condition. Likewise, task n_7 is allocated to *C*(n_4). Task n_8 is allocated to processor *C*(n_8) where its parent task n_6 is allocated because n_8 has the largest ready time from n_6. As the result, result schedule becomes as shown in Fig. 3 (a).



task	est	ect	fred
n_1	0	1	
n_2	1	2	n_1
n_3	1	2	n_1
n_4	1	2	n_1
n_5	1	2	n_1
n_6	2.8	3.9	n_2
n_7	2.9	3.9	n_4
n_8	4.8	5.8	n_6

Figure 3. Allocation of tasks by Darbha's algorithm: (a) task schedule by Darbha's algorithm, (b) properties of nodes used by Darbha's algorithm

In Fig. 3 (a), schedule length 5.8 is optimal and the schedule uses 4 processors. In the Figure, each processor is notated as $C(n_i)$ where n_i is the last task allocated to the processor.

Let us check whether the DAG in Fig. 3 (a) satisfies Darbha's Condition. Table. I shows a check process whether each join task in DAG shown in Fig. 1 (a) satisfies the inequality: first, look at the first row that compares the inequality for n_2 and n_3 in join task n_6. Since *est*() of task n_2 and n_3 are the same, Darbha's Condition $\tau_2 \geq c_{2,6}$ is checked. Because execution time of task n_2, which is 1, is larger than the communication time from n_2 to n_6, the inequality is satisfied. Likewise, task n_2 satisfies Darbha's Condition as shown in the second row of table 1. In case of join task n_2, since start time of the largest ready time parent task n_6 is smaller than that of n_7, the second condition of Darbha's Condition is checked. The condition is satisfied as shown in the third row of Table I.

TABLE I. CHECK FOR DARBHA'S CONDITON

Join task	If condition	condition	value	result
n_6	$est(n_2) \geq est(n_3)$	$\tau_5 \geq c_{2,6}$	$1 \geq 0.9$	satisfy
n_7	$est(n_4) \geq est(n_5)$	$\tau_4 \geq c_{5,7}$	$1 \geq 0.9$	satisfy
n_8	$est(n_6) \geq est(n_7)$	$\tau_6 \geq c_{7,8} + est(n_7) - est(n_6)$	$1.1 \geq 0.9 + 2.9 - 2.8$	satisfy

Although TDS algorithm generates a shortest schedule given a DAG that satisfies Darbha's condition, it does not mean that merge of any processors increases its schedule length. For example, DAG in Fig. 1 satisfies Darbha's condition. Schedule in Fig. 1 (a) is generated by TDS algorithm. Merging of task n_3 to $C(n_8)$ in the schedule increases completion time of n_6 from 3.9 to 4.1. However, Fig. 4 shows that the merge does not affect completion time of task n_8. From the example, we know that schedule length can be maintained even if the number of processors is reduced in case that the schedule is generated by TDS algorithm.

Fig. 5 (a) shows an example of a DAG where most join tasks satisfy the second condition of Darbha's Condition. For join tasks n_8, n_9, and n_{10} in the DAG, start times of their *fpred* tasks are smaller than those of *spred* tasks. TDS algorithm generates a schedule shown in Fig. 5 (b). The number of required processors in schedule in Fig. 5 (b) can be reduced to a schedule that uses 2 processors and has the same schedule length. By merging $C(n_5)$ to $C(n_{10})$ and merging $C(n_7)$ to $C(n_9)$ as shown in Fig. 5 (b). The schedule is modified to a schedule that uses two processors as shown in Fig. 5 (c).

In a DAG that does not satisfy Darbha's Condition, average communication cost is larger than average

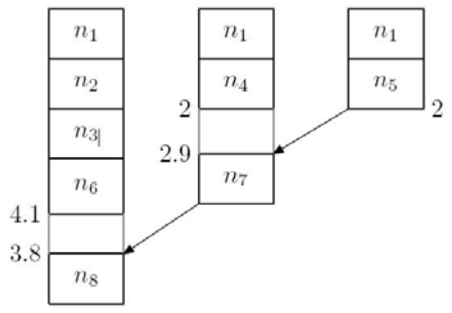


Figure 4. Merging task n_3 to cluster $C(n_8)$

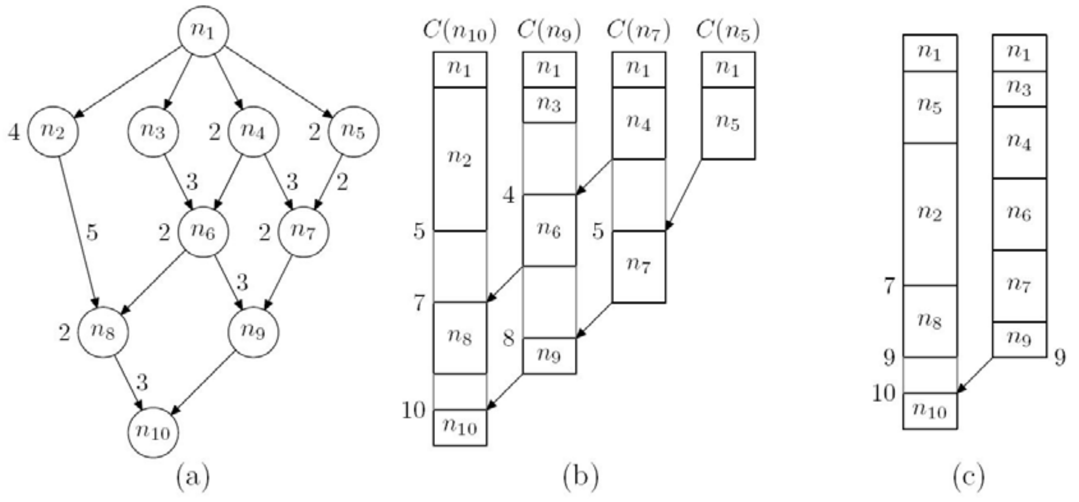


Figure 5. (a) DAG where join task satisfies Darbha’s condition and (b) schedule by Darbha's algorithm. (c) Reduced schedule by merging $C(n_5)$ to $C(n_{10})$ and $C(n_7)$ to $C(n_9)$

computation cost in general, and allocating two or more parent tasks of a join task in the same processor can reduce the schedule length. This paper excludes the conditions where Darbha's Condition does not satisfy. We concentrate on DAGs where Darbha's Condition is satisfied and derive conditions where merge between two processors are allowed while the schedule length is maintained.

If DAG in Fig. 1 (a) is modified such that weight of task n_6 is larger than 1.8, merging like Fig. 4 increases its completion time and the merging is not acceptable. Also, merging $C(n_5)$ to $C(n_7)$ in Fig. 3 increases its schedule length. State of Fig. 5 (b) is highly acceptable, because any merging of a parent task to a processor where its child join task is allocated does not increase the schedule length. However, if the DAG modified to a DAG in Figure 6 by reducing the weights of edges $e_{4,7}$ and $e_{5,7}$ to 2 and 1, respectively, any merging increases the schedule length. Note that modified weights of the two edges make the second of Darbha's Condition unsatisfied.

In order to prevent its schedule length from increasing when two processors are merged, there should be an empty slot in at least one processor. Let express it more formally: given two processor $C(n_i)$ and $C(n_j)$, for task $n_k \in C(n_j) - C(n_i)$, if size of empty slot after $st_{C(nj)}(n_j)$ is equal to or greater than τ_k, there is possibility that merging does not increase the schedule length.

When processor $C(n_j)$ is merged to $C(n_i)$, if sum of empty slot in $C(n_i)$ is equal to or greater than weight sum of tasks that is in $C(n_j)$ but is not in $C(n_i)$, then it is possible that merging does not increase the schedule length. Since weight sum of tasks in $C(n_j) - C(n_i)$ is $\sum_{nk \in C(nj) - C(ni)} \tau_k$ and sum of empty slot in $C(n_i)$ is $ct_C(n_i) - \sum_{nk \in C(nj)} \tau_k$, a merge condition that has a possibility of fixed schedule length is as follows:

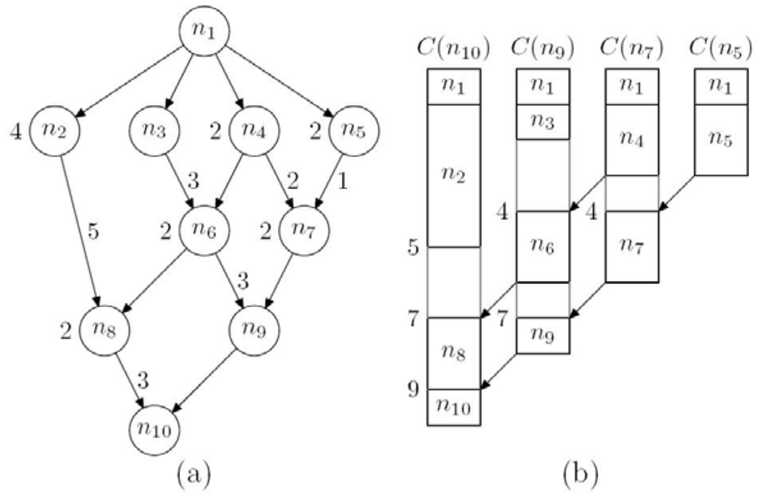


Figure 6. (a) Modified DAG of Fig. 5, (b) schedule by Darbha's algorithm

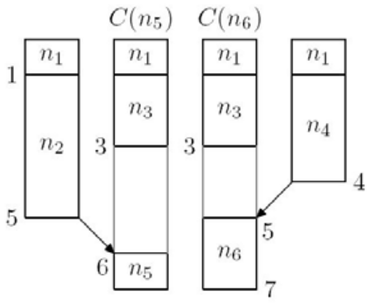


Figure 5. An example where Condition 2 is not suitable

$$\sum_{n_k \in C(n_j)-C(n_i)} \tau_k \le ct_{C(n_i)}(n_i) - \sum_{n_j \in C(n_i)} \tau_j . \tag{1}$$

By the way, although equation 1 is satisfied, merging could increase its schedule length, because there could be a part of the empty slot which does not filled with tasks in $C(n_j)$. In the Fig. 7, when $C(n_6)$ is merged to $C(n_5)$, size of empty slot in $C(n_5)$ is 3. The empty slot is sufficiently large to include task n_6 and it satisfies equation 1. Unfortunately, since $st_{C(n6)}(n_6) = 5$, upper empty slot with size 2 cannot be used and merge will increase the schedule length.

In order to prevent such increase, we consider a tighter condition that considers cases like Fig. 7. Right side of equation 1 is the total amount of empty slot in processor $C(n_i)$. More exact amount of right side is the amount of empty slots that can be filled by merged tasks. When processor $C(n_i)$ and $C(n_j)$ merge, the earliest start time of tasks to be merged is expressed as $\min_{n_k \in C(n_j)-C(n_i)} st_{C(n_j)}(n_k)$. Merging $C(n_i)$ and $C(n_j)$ could increase its schedule length if the sum of empty slot after the start time is smaller than the weight sum of tasks in $C(n_j)$ - $C(n_i)$. Thus the condition can be written as follows:

Condition 2. *If processor $C(n_i)$ and $C(n_j)$ are merged, following condition is required in order to keep the schedule length.*

$$\sum_{n_k \in C(n_j)-C(n_i)} \tau_k \le e_{C(n_i)}\Big(\min_{n \in C(n_j)-C(n_i)} st_{C(n_j)}(n_k)\Big), \tag{2}$$

where $e_{C(ni)}(t)$ means the size of empty slots in processor $C(n_i)$ after time t.

Size of empty slots in a processor can be calculated by substituting weight sum of tasks from completion time of the last task in the processor. Thus the size of empty slot $e_{C(ni)}$ in processor $C(n_i)$ is as follows:

$$e_C = ct_{C(n_i)}(n_i) - \sum_{n_j \in C(n_i)} \tau_j . \tag{3}$$

In some cases, we need to calculate size of empty slots after a time t, in order to check whether the schedule length increases after merging two processors. The size of empty slots in processor $C(n_i)$ after time t is expressed as follows:

$$e_{C(n_i)}(t) = ct_{C(n_i)}(n_i) - t - \sum_{n_j \in C(n_i), st_{C(n_j)}(n_k) < t} \min(\tau_k, ct_C(n_k) - t). \tag{4}$$

In Figure 3, merging cluster $C(n_5)$ or $C(n_3)$ to $C(n_8)$ satisfies Condition 2, while merging $C(n_7)$ to $C(n_8)$ not satisfying the Condition.

Computing the total size of empty slots after a specific time is calculated by summing up intervals after completion time of a task and stat time of the next task in the processor. Thus if C is the number of tasks in a processor, time complexity for Condition 2 is $O(C)$ steps. Possibility of merge can be computed more exactly using an algorithmic method with the same time complexity. The method checks whether the completion time of the last task increases by simulating merge of two processors based on start time of their tasks. It is similar to merge process of merge sort which compares sameness and start time of each task in two processors in top-down direction. If two tasks are the same, they are considered as one task. Otherwise, task with smaller start time is allocated prior to the other. If two tasks have the same start time, there are two options: one option is to give higher priority to task that has the greater completion time; the other option is to give higher priority to task that has higher outgoing degree. Figure 8 shows the flow of the steps.

Unfortunately, maintaining the completion time of the last task in a cluster does not guarantee its schedule length. For example, when processor C_i and C_j are merged, let start time of n_j in C_j be t_j and start time of n_j after

```
Merge_Check($C_i$, $C_j$)
   // $C_i$ is larger cluster than $C_j$.
   // $n_i$ and $n_j$ : the start tasks in $C_i$ and $C_j$, respectively.
   ct = 0;
   while($\exists n_i \in C_i$ and ($\exists n_j \in C_j$)
      if ($n_i$ = $n_j$)
         ct ← max(ct, $st(n_i)$) + $\tau_i$;
         $n_i$ and $n_j$ ← the next tasks in each processors;
      else if ($st_{Ci}(n_i) \leq st_{Cj}(n_j)$)
         ct ← max(ct, $st_{Ci}(n_i)$) + $\tau_i$;
         Select following task as $n_i$ in $C_i$;
      else
         ct ← max(ct, $st_{Cj}(n_j)$) + $\tau_j$;
         Select following task as $n_j$ in $C_j$;
      endif
   endwhile
   while ($\exists n_j \in C_j$)
      ct ← max(ct, $st_{Cj}(n_j)$) + $\tau_j$;
      Select following task as $n_j$ in $C_j$;
   endwhile
   return ct ≤ $ct(C_i)$
End
```

Figure 6. An algorithm that checks whether merge of two clusters is allowed

merging to C_i be changed to t_i' which is greater than t_j. If n_j sends a message to task in the other processor, the arrival time of the message is delayed by the merge and the entire schedule length could increase. Thus Condition 2 or algorithm `Merge_Check()` in Figure 8 is recommended to be used for guide rather than deciding schedule length.

V. Proposed Algorithm

In the paper, we propose a post-processing algorithm that reduces required number of processors through processor merges, which does not increases schedule length. Algorithm `Schedule`(G) in Figure 9 shows structure of the proposed algorithm, where G is an input DAG. The result of the algorithm is schedule P_2 a set of processors, where each processor is a set of tasks with start time. Basic criteria for a result schedule P_2 is the number of processors, where smaller number means better performance of the algorithm. In algorithm `Schedule()`, `Darbha's_schedule(`G`)` receives DAG G as an input and produces a schedule P_1 as output [6]. If DAG G satisfies Darbha's Condition, result schedule P_1 has the smallest length.

A. Reducing the Number of Processors

Algorithm `Cluster_merge`(G, P) in Figure 10 reduces the number of required processors in schedule P generated by the previous scheduling algorithm. A schedule generated by TDA algorithm can be an input. Since all processor pairs can be merged, there are $p(p-1)/2$ possible merge cases, where p is the number of processors.

Rather than merging a processor pair as soon as it is found, `Cluster_merge`() starts merge after finding all possible merge pairs in order to prevent being trapped in local optima and to reduce time complexity. Possible

```
Schedule($G$)
   // input DAG $G$
   $P_1$ = Darbha's_schedule($G$);
   $P_2$ = Cluster_Merge($G$, $P_1$);
   return($P_2$);
```

Figure 7. Main flow of scheduling algorithm that uses the proposed merge after an optimistic scheduling algorithm

```
Cluster_Merge(G, P)
   // G : input DAG
   // P : a schedule consists of a set of processors
   While(length(P) does not increase)
     $E_H \leftarrow \phi$;
     H = (P, $E_H$);
     For(any processor pair ($C_i$,$C_j$) in H)
       If(($C_i$,$C_j$) satisfies condition 2)
          add edge ($C_i$,$C_j$) to $E_H$;
       endif
     endfor
     M = Select_Pairs_in_Graph(H);
     P = modified schedule by merging all pairs in M;
   endwhile
   return(previous P);
```

Figure 8. An algorithm that merges processors. The algorithm maintains schedule length and the number of processors are reduced.

merge is decided by Condition 2. If two processors C_i and C_j are possible to merge, a processor pair (C_i,C_j) can be expressed as an edge in a graph $H = (P, E_H)$, where P is a set of processors and E_H is a set of undirected edges between processors. `Cluster_merge`() selects a subset M of E_H which is an edge cover and they are not adjacent each other. After selecting M, `Cluster_merge`(R) does merge for all elements in M. That is, two processors C_i and C_j are merged for each element (C_i,C_j) in M. The merging reduces a set of processors in schedule P to a new one that has smaller number of processors. Such selecting processor pairs and merging them are repeated until the schedule length is maintained. `Cluster_merge`(R) returns the previous schedule P rather than the last schedule P. The reason is that the last schedule P has the increased schedule length while the previous schedule P has the minimum schedule length. Function `Select_Pairs_in_Graph`(H) returns a set of pairs M.

While maintaining schedule length, `Cluster_Merge`() in Figure 10 does not dramatically reduce the number of processors so much. If we change the objective to minimizing the computing power (schedule length x the number of processor) as used in Cloud computing systems, slightly changed algorithm can reduce the computing power more by decreasing the number of processors highly with a little cost of schedule length increase. The modification is to change break condition of do-while iteration from maintaining schedule length to checking existence of mergeable processor pair. `Cluster_Merge_MP`(P) in Fig. 11 is the modified algorithm by changing the iteration condition and reduces computing power than `Cluster_Merge`() does.

B. Selecting Merge Pair

In the paper, we express merging two processor C_i and C_j as a 2-tuple (C_i,C_j). Since a result of merge is not affected by order of processors, (C_i,C_j) is considered as same as (C_j,C_i). Condition 2 is used to decide whether any processor pair can be merged. Order of merge in the pairs decides the number of processors in result schedule. For example, the number of possible merge in Figure 5 (b) is 4(4-1)/2 = 6. Among them, pairs that satisfy Condition 2 are ($C(n_5)$,$C(n_7)$), ($C(n_5)$,$C(n_9)$), ($C(n_5)$,$C(n_{10})$), and ($C(n_7)$,$C(n_9)$). Schedule length does not increase after any merge of above 4 pairs. However, merging ($C(n_5)$,$C(n_7)$) or ($C(n_5)$,$C(n_9)$) prevent following additional merge and result in schedule with 3 processors. On the other hand, merging which start with ($C(n_5)$,$C(n_{10})$) can

```
Cluster_Merge_MP(P)
   // P : a schedule consists of a set of processors
   While(M ≠ $\phi$)
     $E_H \leftarrow \phi$;
     For(any processor pair ($C_i$,$C_j$) in H)
       If(($C_i$,$C_j$) satisfies condition 2)
         add edge ($C_i$,$C_j$) to $E_H$;
       endif
     endfor
     Select_Pairs_in_Graph(H, M);
     Reduce H by merging processor pairs in M;
   endwhile
```

Figure 9. An algorithm that merges processors while there is any processor pair that satisfies merge condition

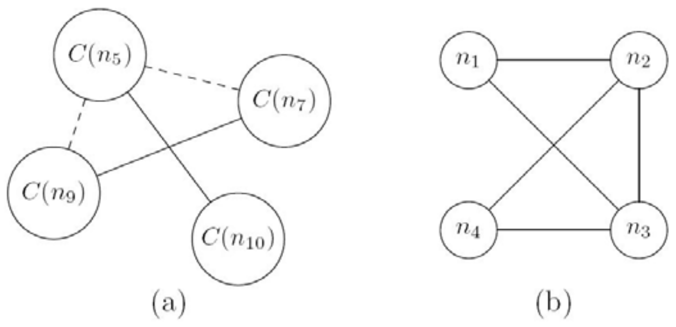


Figure 10. Graphs that represent processor merges : (a) merge graph from Fig 5 (b), and (b) another merge graph

generate a schedule that uses the smallest number of processors.

TABLE II. CHARACTERISTICS OF DAGS AND INITIAL ALLOCATIONS USING DARBHA'S SCHEDULING ALGORITHM

The number of tasks	***The number of edges***	***Schedule length***	***The number of processors***
101	211.37	117.23	34.25
201	423.89	129.52	68.15
501	1060.35	144.82	170.73
1001	2121.53	156.05	341.83
2001	4246.64	165.81	682.29
5001	10608.79	180.76	1701.65
10001	21222.58	189.54	3412.58

The processor pairing problem can be simplified as a graph problem. Fig. 12 (a) is an example of representing processor pairing in the form of edge covering, where a processor is represented as a node and a mergeable pair is represented as an edge. In the graph, desirable merges are ($C(n_5)$,$C(n_{10})$) and ($C(n_7)$,$C(n_9)$). The merges generate a schedule with two processors as shown in Fig. 5 (c).

There could be a doubt that target merge pair is not adjacent processors but any two processors. The reason is that mergeability has no relation with existence of communication between two processors. For example, an optimal merge from Fig. 5 (b) is shown in Fig. 5 (c), where there is no communication between merged processors $C(n_{10})$ and $C(n_5)$. In fact, consideration factors for merge are the size of empty slot and effect on other tasks.

In order to minimize the number of processors, the algorithm should select non-adjacent edges as many as possible. For example, it is better to select solid edges rather than dotted edges in Fig. 12 (a). In other words, we should find a set of edges that covers vertices in the graph as much as possible. Such problem is known as edge cover [1]. In order to find the maximum edge cover given graph H, the proposed algorithm adapts maximum matching algorithm proposed by Edmonds [11]. Time complexity of Edmonds' algorithm is known as $O(VE)$, where V is the number of vertex and E is the number of edges.

C. Time complexity

Major part of the proposed algorithm `Cluster_Merge()` is repetition of finding merge pairs and doing merge. Since the algorithm finds edge cover, the number of processors reduced to half. Thus the number of repetition is at most $\log P$, where P is the number of processors. Constructing a graph of processors H takes $O(P^2C)$ steps, where C is the number of tasks in a processor. Thus it can be rewritten as $O(PV)$. Finding edge cover takes $O(VE)$ steps. Merging each pair takes $O(C)$ steps and the merging runs $O(P)$ times. Thus merging takes $O(V)$ steps. After all, time complexity of the algorithm is $O((PV+VE+V)\log P)$ or $O(VE\log P)$ in short.

VI. EXPERIMENTS AND ANALYSIS

A. Experiment overview and environments

In order to compare performance of algorithms, we generated random graphs that satisfy Darbha's Condition as shown in Table II. Table II shows specifications of schedule after applying TDS algorithm from randomly generated DAGs, which contains 101, 201, 501, 1001, 2001, 5001, and 10001 tasks respectively. 100 DAGs are generated for each size. Randomly generated DAG can have multiple exit tasks with high probability. In order to make a single exit task for each DAG, a dummy node is generated and new edges from the exit tasks to the dummy task are generated to connect them. As the result, most DAG has one more task. The number of edges is determined such that degree of a node is randomly distributed between 3 and 5. Thus the number of edges is about double of the number of nodes. Node weights are uniformly distributed integer values between 6 and 10, and edge weights are uniformly distributed integer values between 4 and 8. Reason that node weights are greater than edge weights is to easily generate DAGs that satisfy Darbha's condition.

TABLE III. FOUR CASES OF DECISIONS ACCORDING TO TASK PRIORITY AND MERGEABILITY CHECK

Merge-ability check / **Priority of tasks**	***Condition 2***	***Function Merge_Check()***
Randomly	Case R.2	Case R.M
Higher out-degree	Case O.2	Case O.M

For each DAG, TDS algorithm generates a schedule, and `Schedule`(*G*, *P*) runs the merge algorithm for the schedule. The experiment considers two parameters and each parameter has two cases. Thus 4 cases are considered as shown in Table. III.

Table IV shows results of `Schedule()` in Fig. 9 by applying above 4 cases for DAGs in Table II. Since schedule length for all cases are the same, Table IV shows the number of reduced processors. `Merge_Check`() function generates a little better results than Condition 2 does. Priority to higher out-degree task contributes more than random selection when merging two processors.

TABLE IV. CHARACTERISTICS OF DAGS AND INITIAL ALLOCATIONS USING DARBHA'S SCHEDULING ALGORITHM

The number of tasks	**Case R.2**	**Case R.M**	**Case O.2**	**Case O.M**
101	94.2%	93.8%	92.8%	92.3%
201	96.0%	95.4%	95.8%	95.1%
501	95.2%	95.3%	95.0%	95.7%
1001	96.3%	96.3%	95.9%	96.5%
2001	96.8%	96.0%	96.7%	96.1%
5001	97.4%	97.0%	98.0%	96.8%
10001	97.5%	97.6%	96.7%	97.0%
average	96.2%	95.9%	95.8%	95.7%

The result in Table. 4 is affected by an iteration condition in `Cluster_Merge`() of Fig. 10 that stops the iteration if the next merge increases schedule length. Thus the reduction amount in the number of processors is not so remarkable. Let us change the iteration stop condition such that until there is no processor pair to be merged. That is, results by allocating task with higher out-degree prior and by using function `Merge_Check()`has better schedule length at the cost of smaller reduction in the number of processors. Table. V shows that applying function `Cluster_Merge_MP()` increases schedule length a little but reduces the combined cost much compared to function `Cluster_Merge()`. The number of processors is reduced to 11.95% ~ 21.06% while schedule length is increased to 3.05% ~ 6.89%. Let us define computation cost is a product of schedule length and the number of processors. The proposed merge algorithm generates schedules of which computation cost is 84.3% ~ 91.2% compared to the schedule by `Cluster_Merge()`.

VII. CONCLUSIONS

The paper proposes a post-process scheduling algorithm that reduces the number of processors of given schedule that is generated from TDS algorithm where input DAG satisfies its optimality condition. The proposed algorithm expresses mergeable processor pairs as graph and applies edge cover in order to maximize the merge effect. If schedule length should be maintained, the proposed algorithm reduces the number of processors to 92.3% ~ 98.0%. If schedule length could be increased, the proposed algorithm reduces the number of processors to 79.3% ~ 85.9% while reducing the computation cost to 84.3% ~ 91.2% compared to the previous schedule.

TABLE V. CHARACTERISTICS OF DAGS AND INITIAL ALLOCATIONS USING DARBHA'S SCHEDULING ALGORITHM

The number of tasks	**Case R.2**		**Case R.M**		**Case O.2**		**Case O.M**	
	sl	***nop***	***sl***	***nop***	***sl***	***nop***	***sl***	***nop***
101	105.3%	80.9%	105.6%	80.2%	105.2%	80.8%	104.9%	80.4%
201	107.0%	79.3%	106.3%	79.6%	106.8%	79.8%	105.8%	79.9%
501	105.7%	80.8%	105.7%	80.9%	105.2%	80.9%	105.3%	81.0%
1001	107.3%	81.8%	107.0%	81.4%	107.0%	81.9%	106.9%	81.7%
2001	106.7%	82.7%	106.3%	82.8%	106.7%	82.8%	105.9%	82.9%
5001	106.1%	83.7%	105.8%	84.7%	105.9%	83.8%	105.7%	84.9%
10001	106.4%	84.4%	106.1%	85.9%	106.2%	84.4%	106.0%	85.8%

The number of tasks	**Case R.2**		**Case R.M**		**Case O.2**		**Case O.M**	
	sl	***nop***	***sl***	***nop***	***sl***	***nop***	***sl***	***nop***
average	106.4%	81.9%	106.1%	82.2%	106.1%	82.1%	105.8%	82.3%

sl means schedule length and ***nop*** means the number of processors

ACKNOWLEDGEMENTS

This paper was supported by Research Fund, Kumoh National Institute of Technology.

REFERENCES

[1] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, ch. A5.2, pp. 238-241. W.H. Freeman and Company, 1979.

[2] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," SIAM Journal on Computing, vol. 19, pp. 322-328, April 1990.

[3] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," Journal of Parallel and Distributed Computing, vol. 16, pp. 276-291, Dec. 1992.

[4] X. Tang and S. T. Chanson, "Optimizing static job scheduling in a network of heteroge-neous computers," in Proceedings of 2000 International Conference on Parallel Processing (29th ICPP'00), (Toronto, Canada), pp. 373-382, Ohio State Univ., August 2000.

[5] K. He and Y. Zhao, "A new task duplication based multitask scheduling method," in Proceedings of the Fifth International Conference on Grid and Cooperative Computing (GCC'06), (Changsha, Hunan, China), pp. 221-227, IEEE Computer Society, 21-23 October 2006.

[6] S. Darbha and D. P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," IEEE Transactions on Parallel and Distributed Systems, vol. 9, pp. 87-95, January 1998.

[7] Y.-K. Kwok and I. Ahmad, "Exploiting duplication to minimize the execution times of parallel programs on message-passing systems," in Proceedings of Sixth IEEE symposium on Parallel and Distributed Processing, pp. 426-433, October 1994.

[8] L. Shen and T.-Y. Choe, "Posterior task scheduling algorithms for heterogeneous computing systems," in VECPAR'06 (high performance computing for computational science, (Rio de Janeiro, Brazil), 10-13, July 2006.

[9] D. Bozdag, F. Ozguner, and U. V. Catalyurek, "Compaction of schedules and a two-stage approach for duplication-based dag scheduling," IEEE Transactions on Parallel and Distributed Systems, vol. 20, pp. 857-871, 2009.

[10] C.-I. Park and T.-Y. Choe, "An optimal scheduling algorithm based on task duplication," IEEE Transactions of Computers, vol. 51, pp. 444-448, April 2002.

[11] Edmonds, J. "Paths, Trees, and Flowers," Canadian J. Math., vol. 17, pp. 449-467, 1965.

AUTHORS PROFILE

Tae-Young Choe, is working as Associate Professor in the Department of Computer Engineering, Kumoh National Institute of Techonology, Gumi City, South Korea. Currently, his research interests are load balancing in Cloud Computing and parallel algorithms using graphic devices.