Improved Cluster Merging by Preemption in Task Scheduling

Tae-Young Choe
Dept.of Computer Engineering, Kumoh National Institute of Techonology
Daehak-ro 61, Gumi, Gyeongbuk, Korea
choety@kumoh.ac.kr

***Abstract*—Most static task scheduling algorithms schedule tasks in non-preemptive mode. Basic reason of such tendency is that preemption invokes overheads and it does not help to minimize its schedule length. We found that preemption helps to reduce the number of processors while the schedule length does not increase. We propose a task scheduling algorithm that applies task preemption. The basic process of the algorithm uses preemption when it merges tasks in two processors into one processor.**

Keyword-task schedule; cluster merge; pre-emption; parallel and distributed systems; cloud computing

I. INTRODUCTION

Cloud computing and grid computing have spread applications that use multiple processors. Such applications are composed of multiple blocks that execute in each processor. Web server is a type of the applications where each block is a request for a web page. In the case no communication exists between blocks and the application can be easily implemented using a thread pool. Another type of the applications is a set of applications where a block should communicate with other blocks. In the case a block is a sequence of codes that use input data and make results, which are transferred to other blocks and are used as input data by the blocks. The data transfer between blocks is considered just as a preference relation if all processors are located in a system and they are connected by a communication switch like a bus and shares a system clock. However, communication cost or time is included as a property of the data transfer if the working processors are located in different system and does not share a system clock like computer clusters, grid computers, or cloud computers. Many traditional computation programs are included in the latter case.

Task scheduling algorithm is to allocate the blocks to processors in order to minimize the completion time of applications. If we consider a task scheduling algorithm that allocates blocks to processors in a cloud computer, the algorithm decides the execution time of applications and performance of the cloud computer. If execution times of blocks and costs communication links are known from the previous execution of computational applications, static task scheduling could finish the application faster than dynamic task scheduling does. Although task scheduling problem is known as NP-complete [1], many static task scheduling algorithms have been proposed, while the algorithms do not allow preemption for task scheduling [2-5]. If preemption is allowed in task scheduling, a block can be partitioned to two or more sub-blocks and other blocks can be executed between the sub-blocks. Unfortunately, most papers that deal with preemptive task scheduling in multiprocessor system are concentrated on satisfying deadline in real-time systems [6-8].

One reason of the unpopularity is that pre-emption does not effect on schedule length. For example, assume that two tasks t_i and t_j are allocated in processor P_i in the order. If t_j preempts t_i in the middle, then the finish time of t_j reduces at the amount of preemption, while the finish time of t_i increases and the completion time of P_i does not reduce at all. Moreover, the preemption invokes context switch overhead [9].

However, reducing the schedule length is not the unique consideration for task scheduling in multi-processor system. The number of used processors determines cost of an application while the schedule length determines the performance of the multiprocessor system. Thus, many task scheduling algorithms are designed in order to reduce the schedule length and to reduce the number of used processors at the same time [10-15]. Methods to reduce the number of used processors can be classified in two groups. One method is to restrict the maximum number of useable processors [10][13], which is suitable when a target system is determined in advance. The other method is to schedule without any restriction and to reduce the number of processors by processor (or cluster) merging [4][16][17]. The second method is suitable if the target system is flexible like a grid or a cluster computing system. We concentrate on the cluster merging because the flexible computing system is more popular.

Cluster merging is an operation that moves all tasks in a cluster to another cluster. Thus cluster merging reduces the number of used clusters. Tasks in a merged cluster are not completely ordered because tasks are ordered by parent and child relations. There is no direct order relation between sibling tasks. Since there should be order in a cluster, topological sorting is applied to tasks in a cluster in general. Unfortunately, ordering between multiple unrelated tasks could invoke unexpected result. For example, assume that clusters P_i and P_j are merged as shown in Fig. 1 (a). If task n_i precedes task n_j in the merged cluster, there is no change in completion time of other tasks as shown in Fig. 1 (b). But if task n_j precedes task n_i, completion time of a child task n_a increase as shown in Fig. 1 (c), which may increase schedule length.

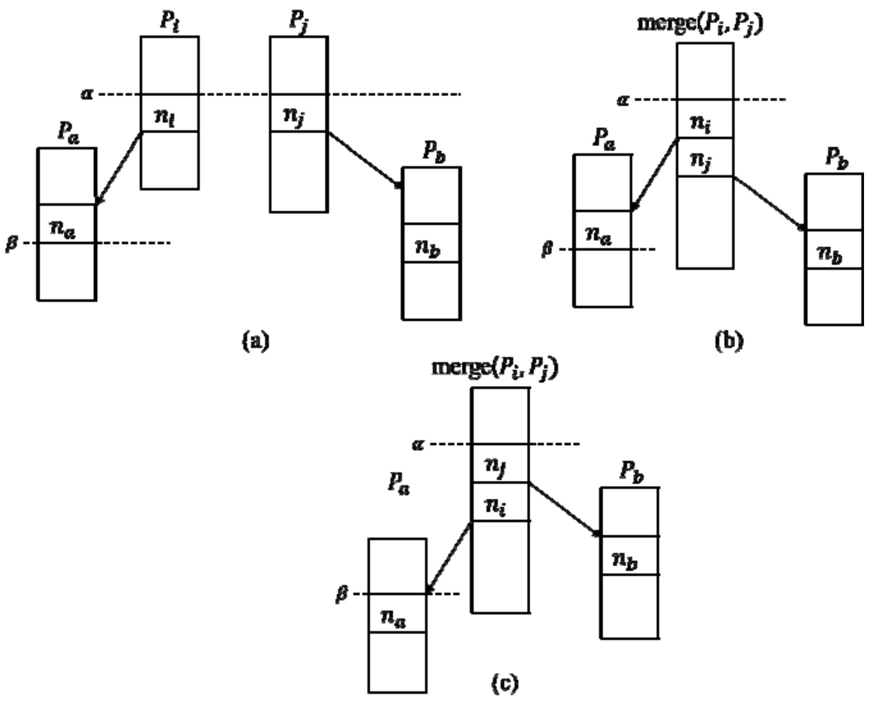

Fig. 1. Ordering effect on cluster merge, (a) before merge, (b) merging that keeps schedule length, (c) merging that increases schedule length

Choe suggested to give higher priority to task that has higher out-degree task [18]. The suggestion come from the intuition that task with higher out-degree has higher probability to effect on schedule length. Also the experimental results show that the suggestion has better performance compared to random selection within the range of error.

Instead of such intuitive suggestion, we discovered a property by which a task should precede the other task. The property can be intuitively explained from Fig. 1. When a message from task n_j arrives to cluster P_b, there is an interval before task n_b starts, which means that execution of task n_j can be delayed by the amount of the interval. But task n_i does not give such interval to cluster P_b. Thus task n_i should finish its execution before task n_j finishes in order to minimize the delay of child tasks. Based on the property, we suggest a condition that cluster merge does not increase schedule length. Since the condition is stricter than general case, we also suggest an algorithm that determines the merge property more exactly.

The rest of the paper is organized as follows. Section 2 defines terms and objectives. Section 3 presents a condition and a decision algorithm where cluster merging does not increase the schedule length. Section 4 shows a frame algorithm that uses the merge operation and an example schedule. Finally the paper concludes in Section 5.

II. TERMINOLOGIES

A *task* is a block of contiguous codes. A task has a size which means the number of instructions to be executed or an execution time. In the paper, the task size is used as the execution time. A task can have a relation with another task. The relation could be a precedence restriction or a data transfer from that task to this task. For example, if task n_a should be executed before task n_b starts, there is a relation from n_a to n_b. A *node* is an entity and an *edge* is a relation between two nodes in a graph. A task is mapped to a node in a graph and they are notated as n_a. A relation is mapped to an edge in a graph and they are notated as $e_{a,b}$ if the edge connects task n_a and n_b. If an application or a module makes a result and terminates in a finite time, it can be represented as a graph, especially directed acyclic graph (DAG) where edges are directed. All edges are assumed to be directed. Fig. 2 shows an example of DAG where each node and edge has weight. Weight of an edge means communication cost between tasks if the tasks locate in different processors. One or more edges are attached to a node. The number of attached edges in a node is called *degree* of the node. *Out-degree* of a node is the number of attached edges that start from the node. *In-degree* of a node is the number of attached edges that arrive in the node. A task that has zero in-degree is called *entry task* and a task that has zero out-degree is called *exit task*.

Task n_1 is the entry task and n_{12} is the exit task in Fig. 2. For generality, we assume that there is an entry task and an exit task in a DAG in order to easily compute completion time of an application.

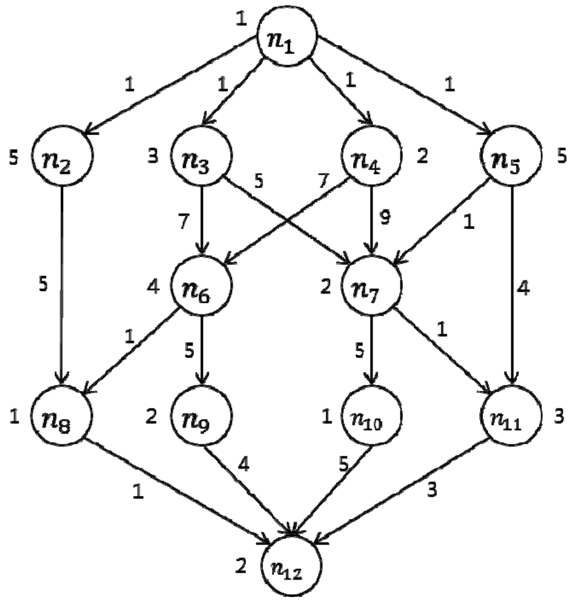

Fig. 2. An example of DAG

Two tasks have relation by an adjacent edge. If there is an edge $e_{a,b}$ from task n_a to n_b, n_a is a *parent* task of n_b and n_b is a *child* task of n_a. In Fig. 2, task n_3 is parent task of task n_7 and n_6 is child task of task n_4. If a task has two or more child tasks, it is called a *fork* task. If a task has two or more parent tasks, it is called a *join* task. Given a task n_a, *PRED*(n_a) is the set of parent tasks of n_a, that is,

$$PRED(n_a) = \{n_b \mid e_{b,a} \in E\} \cdot$$

SUCC(n_a) is the set of child tasks of n_a, that is,

$$SUCC(n_a) = \{n_b \mid e_{a,b} \in E\} \cdot$$

$st_{Cs}(n_a)$ is the start time of task n_a allocated in cluster C_s and $ct_{Cs}(n_a)$ is the finish time of task n_a allocated in cluster C_s. τ_a is the weight of task n_a. If task n_a is not preempted, $ct_{Cs}(n_a) = st_{Cs}(n_a) + \tau_a$. Computation of start time is introduced in Chapter 3.

Task scheduling is an allocation of tasks (nodes) to processors where each task has its *start time* and *finish time* in its processor. An entry task has zero start time because it does not need a message from parent task. If a task has a parent task, it should wait until message from the parent task arrives. If the parent task is allocated to the same processor where the task is allocated, the message communication time is assumed to be zero because it is an intra-processor communication and the size is negligible compared to inter-processor communication. Otherwise the child task should wait until the parent task finishes and the result of the parent task arrives to child task through communication link. A schedule is a set of clusters (or processors) where each cluster is a set of pairs (task, start time). *Schedule length* is a finish time of the exit task in the schedule.

Task scheduling problem is to find an algorithm that generates a schedule with the shortest schedule length. A task can be duplicated in the multiple processors in order to reduce its schedule length.

III. Merge Conditions and Decision Function

A. *Cluster Merging*

A cluster is mapped to a processor in task scheduling. Thus if a task is an element of a cluster, the task is allocated in the corresponding processor. Cluster merging is to combine two clusters and to make a new cluster, which is a union of the two clusters. If cluster merging is applied to a task scheduling, overall process of the task scheduling is as follows:

1) *Initial schedule:* it can be any schedule. For example, a schedule can be constructed such as one task in one cluster. Or any previous task scheduling algorithm can be used in the step.

2) *Cluster merging*: two clusters are merged repeatedly until some pre-defined conditions are satisfied. An example of the pre-defined condition is the maximum available number of processors.

If an initial schedule is one task in one cluster, results of merge are highly un-expectable. Thus any previous task scheduling algorithm that has no restriction on the number of processors is preferred than one task in one cluster initialization. One example of such algorithm is TDS (Task Duplication based Scheduling) algorithm proposed by Darbhan and Agrawal [11]. After initial schedule being decided, clusters are merged until a condition is satisfied. There are two types of merging stop conditions: one is the number of available processors; another is that merge does not increase schedule length. If an application runs in a cluster computer or in a parallel computer where the number of processors is predefined, the number of clusters should be equal or smaller than the predefined number of processors. In the case, cluster merging repeats until the number of clusters is equal or smaller than the predefined number.

If an initial scheduling algorithm generates a schedule with the shortest schedule length and the length should be maintained, cluster merging continues while it keeps the schedule length. TDS algorithm is one of the task scheduling algorithms that generate the shortest length schedule if input DAG satisfies a condition. Roughly speaking, the condition means that cluster merging does not reduce schedule length if each parent task of a join task constitutes its cluster. For example in Fig. 2, join task n_7 has 3 parent tasks n_3, n_4, and n_5. Then there are three clusters $C(n_3)$, $C(n_4)$, and $C(n_5)$, where any cluster merging does not reduce the start time of task n_7. Although cluster merging does not reduce schedule length, it still reduces the number of used processors.

B. Preemption

In a cluster, some tasks are in ordered relation and others are not. Only if they are scheduled to satisfy topological sorting, the application works correctly. Unfortunately, ordering between unrelated tasks can make different schedule length as shown in Fig. 1. In order to order tasks correctly, a scheduler need to know more information for task order. We apply time gap of a task such as the earliest start time and the latest start time [11, 14].

Also we focus on preemption on tasks. Although task preemption reduces the schedule length for some restricted environment [19], it helps to keep schedule length during cluster merging. For example, Fig. 3 is a schedule of DAG in Fig. 2 by TDS algorithm. The DAG satisfies Darbha's condition and there is no cluster merge that can reduce schedule length in Fig. 3. If cluster $C(n_{12})$ and $C(n_{10})$ are merged by force, the result of merge increases the schedule length to 22 as shown in Fig. 4.

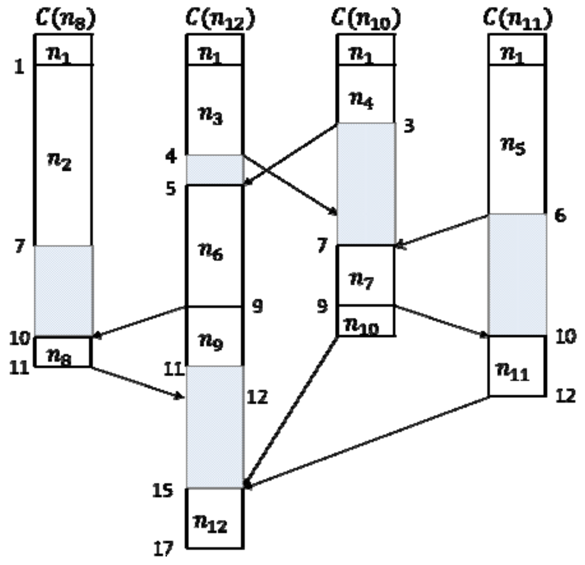

Fig. 3. Schedule of DAG in Fig. 2 by TDS algorithm

Task n_6 and n_7 have edges to child tasks n_8 and n_{11} in other cluster, respectively. It is clear that ordering of the tasks effects the start times of the child tasks. Fig. 4. show that locating task n_6 before task n_7 delays start time of task n_7 and start time of task n_{11}. The delay propagates to the exit task n_{12} which increases schedule length up to 22. Unfortunately, exchanging order of task n_6 and task n_7 does not remedy the increment of schedule length. If task n_7 is allocated before task n_6 in cluster $C'(n_{12})$, task n_7 finishes at time 9, task n_6 finishes at time 13, and start time of task n_{12} becomes 16. Although the amount of delay is reduced, the increment of the schedule length denies cluster merge.

By applying task preemption, the schedule length is preserved as shown in Fig. 5. By allocating task n_6 before task n_7 and preempting n_6 with n_7, start time of task n_{11} is preserved. Also delay of task n_6 by the preemption does not affect start time of the exit task n_{12}.

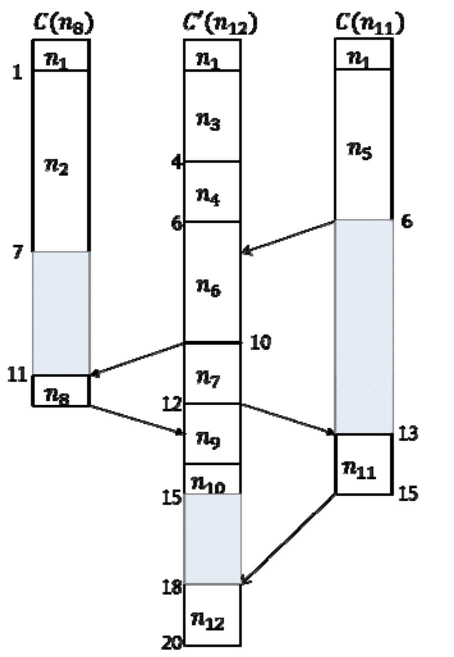

Fig. 4. Merge of cluster $C(n_{10})$ and $C(n_{12})$ from schedule in Fig. 3

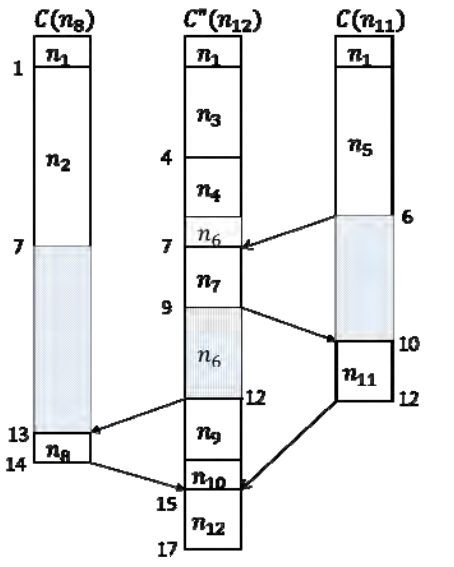

Fig. 5. Merge of cluster $C(n_{10})$ and $C(n_{12})$ using preemption from schedule in Fig. 3

Now, we derive a merge condition that indicates whether cluster can be merged without increasing the schedule length. Given an initial schedule, there are some clusters where tasks are allocated. Before deciding preemption, some properties of tasks should be computed. Start time $st_{Cs}(n_a)$ and *slack time* of task n_a in cluster C_s are such properties. Slack time is an interval between the start time and a *closing start time* of task n_a. Closing start time of task n_a is the latest start time by which start times of its child tasks are not delayed. If n_a start after the closing start time, start times of some its child tasks are delayed and the schedule length could increase as the consequence. Closing start time of task n_a in cluster C_s is notated as $cst_{Cs}(n_a)$.

Start time of a task is 0 in the case of an entry task. Otherwise, it is determined by message arrival time from its parent tasks. A task cannot start before the last message arrival time from parent tasks. According to clusters where parent tasks reside, inclusion of inter-processor communication time to start time is determined. If the task and its parent reside in the same processor, the edge weight between two tasks is ignored. Thus the start time $st_{Cs}(n_a)$ of task n_a is computed as follows:

$$st_{C_s}(n_a) = \max_{n_p \in PRED(n_a)} rdy_{C_s}(n_p, n_a), \tag{1}$$

where $rdy_{Cs}(n_p, n_a)$ is a message arrival time from a parent task n_p to task n_a in cluster C_s and it is called *ready time*. We assume that task duplication is allowed. Thus there could be multiple duplicated task n_p in multiple clusters. Since task n_a need just one message from the multiple parent tasks, one message that arrives the earliest is sufficient. If parent task n_p is located in the same processor with task n_a, communication overhead is ignored. Thus ready time from n_p to n_a in C_s is computed as follows:

$$rdy_{C_s}(n_p, n_a) = \begin{cases} \min\limits_{C_p \ni n_p} ct_{C_p}(n_p) & \text{if } C_s = C_p \\ \min\limits_{C_p \ni n_p}(ct_{C_p}(n_p) + c_{p,a}) & \text{otherwise} \end{cases}. \tag{2}$$

Closing start time $cst_{Cs}(n_a)$ of task n_a is computed from a completion time *mct* which is computed from current start times of its child tasks by subtracting communication cost. Since the completion time is computed from all child tasks, it should be the minimum value of completion times derived from the child tasks. By subtracting the size of n_a, the closing start time is computed as follows:

$$cst_{C_s}(n_a) = \min_{n_i \in SUCC(n_a)} (mct_{Cs}(n_a, n_i) - \tau_a), \tag{3}$$

where $mct_{Cs}(n_a,n_i)$ is the last allowed completion time of task n_a in order for all duplications of its child task n_i to start in the current time. Thus *cst* should be the minimum value for all duplications of its child tasks. If child task n_i is allocated to the same cluster as task n_a, communication cost between the tasks is ignored. Thus,

$$mct_{C_s}(n_a, n_i) = \begin{cases} \min\limits_{C_i \ni n_i} cst_{C_i}(n_i) & \text{if } C_s = C_i \\ \min\limits_{C_i \ni n_i}(cst_{C_i}(n_i) - c_{a,i}) & \text{otherwise} \end{cases}. \tag{4}$$

As long as a task n_a starts between $st_{Cs}(n_a)$ and $cst_{Cs}(n_a)$, the schedule length does not change. There could be a case of $st_{Cs}(n_a) > cst_{Cs}(n_a)$, which means that allocating n_a to C_s could increase the schedule length. Let start time of task n_a be t in cluster Cs. Although start time of n_a is delayed to $cst_{Cs}(n_a)$, the schedule length does not increase. We call the value $cst_{Cs}(n_a) - t$ as slack time and notate as $slt_{Cs}(n_a, t)$. That is,

$$slt_{C_s}(n_a, t) = cst_{C_s}(n_a) - t \cdot \tag{5}$$

If time t is the same as $st_{Cs}(n_a)$, it can be abbreviated to $slt_{Cs}(n_a)$. From above values, a condition for merging with preemption is proposed as following theorem.

Theorem (Preemptive merge condition) *Two clusters C_a and C_b are merging into a new empty cluster C_s, where the last task is n_s. Whenever a task n_a in cluster C_a is selected to be merged into C_s, if one of following conditions are satisfied for all selected tasks, merging two clusters does not increase the schedule length.*

$$st_{C_s}(n_a) \geq ct_{C_s}(n_s), \tag{6}$$

$$\max(st_{C_s}(n_a), ct_{C_s}(n_s)) \leq cst_{C_s}(n_a), \tag{7}$$

$$slt_{C_s}(n_s) \geq \tau_a, \text{ and } st_{C_s}(n_a) - st_{C_a}(n_a) \leq slt_{C_a}(n_a). \tag{8}$$

Proof. Following two cases are situations where merging of n_a into Cs does not increase the schedule length:

1) There is no other tasks of C_s in the slot where n_a would be scheduled: in the case, n_a can be scheduled without any delay. The case is expressed by Equation (6).
2) Task n_s overlaps the slot where n_a would be scheduled: in the case, start time of n_a is delayed by the amount of overlap. If the delayed start time of na does not exceed its closing start time, its child tasks can start in time. The case is expressed by Equation (7).

If above cases are not satisfied, scheduling n_a after n_s delays start times of its child tasks and it could increase the schedule length. Thus n_a tries to preempt ns. If task n_s is preempted, completion time of n_s increases up to the amount of τ_a. In order for child tasks of n_s to keep their start time, slack time of n_s should be equal to or greater than τ_a. Thus the left condition of Equation (8) should be satisfied. Also, delay amount of n_a should be equal to or less than its slack time. Thus the right condition of Equation (8) should be satisfied. □

When two clusters C_a and C_b are merged, checking increment of schedule length is processed as follows:

1. Create an empty cluster C_s
2. Select a task n_a that has the smallest start time among tasks in two clusters.
3. If there is no task in two clusters, jump to step 7.
4. Check whether the schedule length increases when task n_a is inserted into cluster C_s.
5. If it does not increase, move n_a to C_s and jump to step 2.
6. If it increases, return “increase”.
7. Since there is no increment, return “no increase” (or “merge-able”).

When task n_a is moved from cluster C_a to cluster C_s with preemption enabled, following process checks whether the schedule length increases:

1. If $C_s = C_a$, return “no increase” (the step can be skipped).
2. If n_a is already in C_s, return “no increase”.
3. Decide increment based on values *st* and *cst* of n_a in C_s.

Consider a method to check increment of schedule length based on *st* and *cst*. Assume that task n_s is just allocated to the end of cluster C_s, and task n_a is about to be merged into cluster C_s.

1. Compute $st_{Cs}(n_a)$ using Equation (1).
2. If $st_{Cs}(n_a) \geq ct_{Cs}(n_s)$, then merge n_a to C_s and return “no increase”.
3. Compute $cst_{Cs}(n_a)$ using Equation (3).
4. If $\max(st_{Cs}(n_a), ct_{Cs}(n_s)) \leq cst_{Cs}(n_a)$, then merge n_a to C_s and return “no increase”.
5. Otherwise, there is possibility of schedule length increment. Process follows:
   A. If $slt_{Cs}(n_s) \geq \tau_a$, then check 5-B, otherwise do not merge and return “increase”.
   B. If $st_{Cs}(n_a) - st_{Ca}(n_a) \leq slt_{Ca}(n_a)$, then n_a preempts n_s and return “no increase”.
   C. Do not merge and returns “increase”.

In short, if one of conditions in step 2, 4, or (5-A and 5-B) is satisfied; merging task n_a to cluster C_s does not increase the schedule length.

IV. An Example

We show an example of the proposed merge condition and merge process using figures from Fig. 2 to Fig. 5. A DAG is given as shown in Fig. 2. Numbered labels mean weight of node or edge. The DAG satisfies the optimality condition proposed by Darbha and Agrawal [11]. In order to process fast, TDS algorithm generates an initial schedule as shown in Fig. 3. Needless to check the optimality condition, any cluster merging does not decrease the schedule length 17. For example, if two clusters $C(n_8)$ and $C(n_{12})$ are merged, then the completion time of the exit task n_{12} would be 19. Also merging of other two cluster $C(n_{10})$ and $C(n_{11})$ increases the completion time of task n_{12} up to 18. Merging cluster $C(n_{10})$ and $C(n_9)$ without preemption increases schedule length up to 20 as shown in Fig. 4. Major reason of the increment is the delay of task n_7, which started at 7 before merging and starts at 10 after merging. On the other hands, task n_6 allocated before task n_7 effects on start time of the exit task n_{12} through task n_8, where message arrival time from task n_8 is too early for task n_{12}.

Consider step to decide whether merging two cluster clusters $C(n_{12})$ and $C(n_{10})$ does not increase the schedule length. An empty cluster $C''(n_{12})$ is created. Since $C(n_{12})=\{n_1, n_3, n_6, n_9, n_{12}\}$ and $C(n_{10})=\{n_1, n_4, n_7, n_{10}\}$, task n_1 is selected from $C(n_{12})$ and is inserted into $C''(n_{12})$. Next, n_1 in $C(n_{10})$ is selected, but it is abandoned because already n_1 is in $C''(n_{12})$. Task n_3 is selected and is checked for effect on its child tasks. Since child tasks n_6 and n_7 will be merged with n_3, they are excluded for consideration. Thus n_3 is allowed to be merged into $C''(n_{12})$. Task n_4 also has child tasks which are all included in merging clusters, and is merged into $C''(n_{12})$. Next, task n_6 is selected. A child task n_9 is included in merging cluster. Since another child task n_8 is not in merging clusters, effect from n_6 to n_8 should be considered. Closing start time $cst(n_6)$ is determined by closing start time of n_8. Since the gap between the arrival time of message from n_8 and start time of n_{12} is 3, $cst(n_8)$ is 13. Thus $cst(n_6)$ is 8. Since $st(n_6)$ is 5 and $ct(n_4)$ is 6, merge is allowed in step 4. Next, task n_7 is considered. Start time $st(n_7)$ is 7 and $ct(n_6)$ is 10. So $cst(n_7)$ should be computed in step 3. Task n_7 has a child task n_{11} which locates outside $C(n_{12})$ and $C(n_{10})$. Since $cst(n_{11})$ is 10, $cst(n_7)$ is 7. Closing start time $cst(n_7)$ is smaller than the max value in step 4. Slack time $slt(n_6)$ is 2 and τ_7 is 2, which makes step 5-B to be tested. Since start time $st_{C(n12)}(n_7)$ is 7, $st_{C(n10)}(n_7)$ is 7, and $slt_{C(n10)}(n_7)$ is 0, Condition in step 5-B is satisfied. Thus n_7 preempts n_6 at time 7 and is merged into $C''(n_{12})$. Following tasks n_9 and n_{10} has only one child task n_{12}. Thus they checks only start time and the schedule length is preserved. As the result of merge, the schedule is changed as shown in Fig. 5.

V. Conclusion

We propose a preemptive cluster merge method for static task scheduling algorithm and merging conditions. The conditions check whether the cluster merging increase the schedule length. If a given DAG satisfies the optimality condition and TDS algorithm generates an initial schedule, the conditions help to reduce the number of used processors. If a random DAG is given, any scheduling algorithm can generate an initial schedule and the condition help to reduce the number of used processors and to reduce the schedule length.

Since the proposed conditions are not commonly accounted condition, frequency and effects should be investigated in detail. We will adapt the proposed condition and steps to random and application DAGs as future works.

ACKNOWLEDGMENT

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

REFERENCES

[1] M. R. Gary and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] T. Megal, R. Sirdey, and V David, "Minimizing Task Preemptions and Migrations in Multiprocessor Optimal Real-Time Schedules," in *IEEE 31st Real-Time Systems Symposium (RTSS)*, pp.37-46, 2010.

[7] K. Bletsas and B. Andersson, "Preemption-Light Multiprocessor Scheduling of Sporadic Tasks with High Utilization Bound," in *IEEE 30th Real-Time Systems Symposium*, pp.447-456, 2009.

[8] Sun Wei, "A Novel Genetic Admission Control for Real-Time Multiprocessor Systems," in *International Conference on Parallel and Distributed Computing, Applications, and Technologies*, pp.130-137, 2009.

[9] A. Silberschatz, G. Gagne, and P. B. Galvin, Operating System Concepts 8th edition, Wiley, 2011.

[10] Amit Agarwal and Padam Kumar, "Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems," in *Proceedings of the 2009 IEEE International Advance Computing Conference (IACC 2009)*, pp.87-93, Patiala, India, 6-7 March 2009.

[11] S. Darbha and D. P. Agrawal, "Optimal Scheduling Algorithm for Distributed-Memory Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol 9, no. 1, pp.87-95, January 1998.

[12] I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Transactions on Parallel and Distributed Systems,* vol. 9, no. 9, pp.872-892, September 1998.

[13] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp.260-274, March 2002.

[14] 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, no. 6, pp.857-871, June 2009.

[15] A. Agarwal and P. Kumar, "Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems," in *2009 IEEE International Advance Computing Conference (IACC 2009),* pp.87-93 Patiala, India, 6-7 March 2009.

[16] Kun He and Yong Zhao, "A New Task Duplication Based Multitask Scheduling Method," *In Proceedings of the Fifth International Conference on Grid and Cooperative Computing (GCC;06),* pp.221-227, 2016.

[17] M. A. Palis, Jing-Chiou Liou, and David S. L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 1, January 1996.

[18] T. Y. Choe, "Task Scheduling Algorithm to Reduce the Number of Processors using Merge Conditions," *International Journal on Computer Science and Engineering,* vol. 4, no. 2, February 2012.

[19] Z. Gu, X. He, and M. Yuan, "Optimization of Static Task and Bus Access Schedules for Time-Triggered Distributed Embedded Systems with Model-Checking," in *Proceedings of the 44th annual Design Automation Conferenc (DAC'07),* pp.294-299, June 2007.