A Multiprocessor Scheduling Algorithm for Low Overhead Fault-Tolerance

Koji Hashimoto Tatsuhiro Tsuchiya Tohru Kikuno
Department of Informatics and Mathematical Science
Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
E-mail:{k-hashi, t-tutiya, kikuno}@ics.es.osaka-u.ac.jp

Abstract

In this paper, we propose a new scheduling algorithm for achieving fault-tolerance in multiprocessor systems. The new algorithm partitions a parallel program into subsets of tasks based on some characteristics of a task graph. Then for each subset, the algorithm duplicates and schedules its tasks successively. Applying the proposed algorithm to three kinds of practical task graphs (Gaussian elimination, Laplace equation solver and LU-decomposition), we conduct simulations. Experimental results show that fault-tolerance can be achieved at the cost of small degree of time redundancy, and that performance in the case of a processor failure is improved compared to a previous algorithm.

1 Introduction

Viewing multiple components as redundancy, it can be said that parallel and distributed systems have the potential for fault-tolerance inherently. Making use of this property, many researchers have developed multiprocessor scheduling algorithms for achieving high reliability on various system models [3, 8, 14]. For example, Gu et al. [6] have investigated formal characterization of fault-secure multiprocessor schedules. A schedule is said to be fault-secure if either the system produces correct outputs for the program or it detects the presence of faults in the system. In their model, a parallel program is composed of a set of tasks and represented by a directed acyclic graph. All tasks have one time unit execution time and communication delays between processors are not taken into account. In [6], some scheduling algorithms are proposed under this model. However, these algorithms can be applied only to a class of tree-structured task graphs. Chabridon et al. [2] have developed a scheduling algorithm which ensures that the program can run correctly if at least one of the processors is operational. Since the fault-tolerance is achieved by rescheduling and re-execution of tasks upon fault detection, considerable decrease in the performance is inevitable even when a single fault occurs. Gong et al. [5] have studied duplication of operations for fault detection. They consider loop iterations called regular loops, which are perfectly nested and contain no branches, and thus their proposed method cannot be applied to any other kinds of programs than regular loops.

In this paper, we propose a new scheduling algorithm for tolerating a single processor failure in multiprocessor systems with a distributed memory architecture, in which processors communicate with each other solely by message-passing. We consider parallel programs represented by a directed acyclic graph with arbitrary computation and communication costs. Duplicating every task of a given program, the new algorithm ensures that the system can complete the program without rescheduling even if a single processor failure occurs. With this scheduling, fault-tolerance can be achieved at the cost of small degree of time redundancy without requiring any additional hardware.

In previous work [7], we proposed a fault-tolerant scheduling algorithm. In the previous algorithm, a parallel program is partitioned into several subsets of tasks at the first phase, and then tasks in each subset are duplicated and scheduled successively. As analyzed in [7], this two-phase scheduling policy can produce good fault-tolerant schedules because it distributes replicated copies of each task among processors appropriately. The previous algorithm divides all tasks equally into $l(=2,3,\cdots)$ subsets.

In this paper, we further focus on the structure of given task graphs, then utilize the structural information for partitioning more directly than the previous algorithm. Specifically, we introduce a notion of *heights* of tasks and propose a new fault-tolerant scheduling algorithm incorporating height-based partitioning. Through simulation studies, we show that the proposed algorithm can achieve better performance than the previous algorithm particularly in the case of a processor failure. Moreover, the running time of the new algorithm becomes much smaller because its structure can be simple using a special property every partitioned subset has.

The remainder of this paper is organized as follows: The system model assumed in this paper is de-

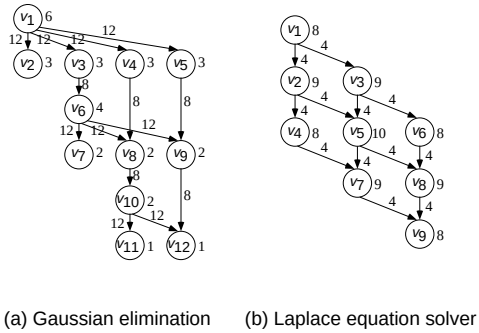


Figure 1: Task graphs.

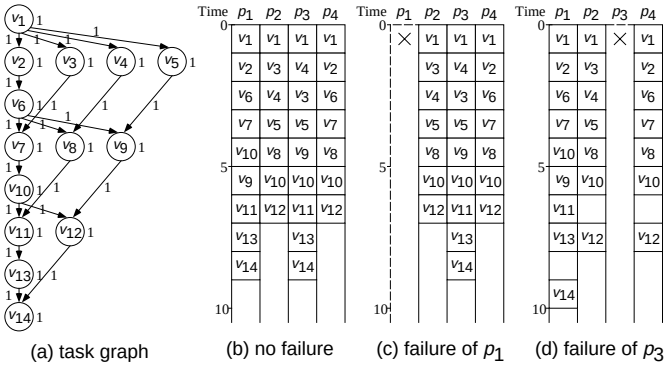


Figure 2: An example of a fault-tolerant schedule.

scribed in Section 2. In the section, the concept of fault-tolerant schedule is also described. The previous scheduling algorithm is explained in Section 3. In Section 4, the proposed scheduling algorithm is described. The results of the simulation studies are shown in Section 5. The paper concludes with Section 6.

2 Preliminaries

2.1 System and Task Model

We consider a multiprocessor system consisting of n identical processing elements (PEs), which runs one application program at a time. All PEs are fully connected with each other via a reliable network. A PE can execute tasks and communicate with another PE at the same time. This is typical with I/O processors and direct memory access. In addition, all PEs are assumed to be fail-stop [12].

A parallel program can be represented by a weighted directed acyclic graph (DAG) $G = (V, E, w, c)$, where V is the set of nodes and E is the set of edges. Each node represents a task v, and is assigned a computation cost $w(v)$, which indicates the task execution time. Each edge $< v, v' > \in E$ from v to v' corresponds to the precedence constraint that task v' cannot start its execution before receiving all necessary data from task v. Given an edge $< v, v' >$, v is called an *immediate predecessor* of v', and v' is called an *immediate successor* of v. Each edge is assigned a communication cost $c(v, v')$, which indicates the time required for transferring the data between different PEs. If the data transfer is done within the same PE, the communication cost becomes zero. In the following, we call such a weighted DAG a *task graph*. Various applications are known to be represented by weighted DAGs. A multitarget tracking algorithm is such an example [13]. Figure 1 shows examples of task graphs. In a task graph, the number adjacent to a node represents the execution time of its corresponding task, and the number on each edge is a communication cost.

We introduce some definitions and terminology as in [7]. For a path in a task graph, its *length* is defined as the summation of task execution times along the path excluding communication delays. The *level* of a task is defined as the length of the longest path from the node corresponding to the task to a node which has no succeeding nodes. In Figure 1(b), for example, the levels of v_4 and v_7 are 25 and 17, respectively.

2.2 Fault-Tolerant Schedule

Fault-tolerant scheduling we discuss here is to produce a schedule with which the system can complete the program even if any single PE failure occurs. We call such a schedule a *fault-tolerant schedule* [7]. The goal of fault-tolerant scheduling is to minimize the schedule length while achieving fault-tolerance. The following example illustrates the basic concept of fault-tolerant schedule.

Example 1 Figure 2(b) shows a schedule for a task graph in Figure 2(a). In this schedule, every task is assigned to at least two different PEs. To distinguish between a task $v \in V$ and its actually scheduled copies, we call the latter the *instances* of v. Now suppose that p_1 failed at time 0 as shown in Figure 2(c). Even in this situation, all remaining instances which are not assigned to p_1 are executed in the same way as the case where no failed PE exists. On the other hand, if p_3 fails at time 0 as shown in Figure 2(d), the instances of v_{12} assigned to p_2 and p_4 cannot be executed at the time when they are originally scheduled. This is because the earliest scheduled instance of its immediate predecessor v_9 cannot be completed. However, if the instances of v_{12} on p_2 and p_4 wait to receive the necessary data from another instance of v_9 on p_1, then all instances scheduled onto healthy PEs can be executed as shown in Figure 2(d).

Note that in Example 1, the execution order of instances on each PE does not change even when a PE fails. Thus, though the finish time of the program is delayed, all instances on healthy PEs can complete their execution without rescheduling. This mechanism can be easily realized by simply checking message arrival at the beginning of each task. In addition, because all tasks are redundantly scheduled to at least two PEs, for any task at least one of its instances is executed. As a result, with such a schedule, the program terminates and all of its tasks are successfully executed even when a PE has failed.

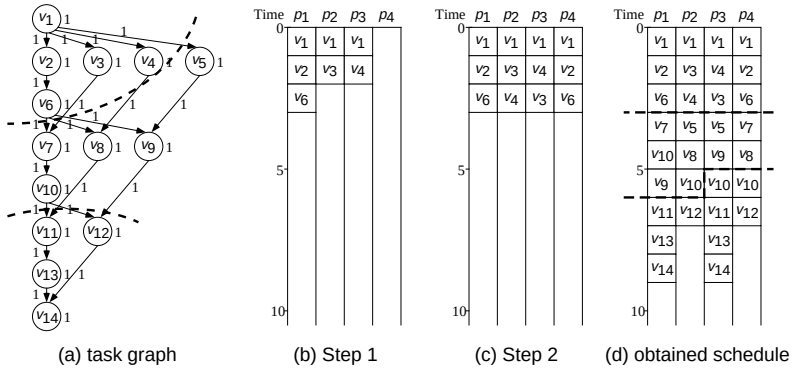


Figure 3: Illustrative example of PHS^3.

3 Our Previous Algorithm

Under the model described in Section 2, we have proposed fault-tolerant scheduling algorithms in our preliminary work [7]. In this section, we explain the previous algorithm.

In [7], we first proposed a set of scheduling algorithms, RSR_1, RSR_2, RSR_3, $\cdots$. Each RSR_k is an extension of Algorithm DSH, which is a non-fault-tolerant scheduling algorithm proposed by Kruatrachue in [9]. Algorithm RSR_k generates a non-fault-tolerant schedule for $n-k$ PEs by applying DSH (Step 1), and then modifies it to be fault-tolerant using n PEs (Step 2). The schedule obtained at Step 1 includes some duplicated tasks because DSH duplicates tasks in order to eliminate communication delays and improve performance. At Step 2, tasks that were not duplicated at Step 1 are duplicated and scheduled.

For achieving better performance, we then proposed Algorithm GRD which integrates RSR_k's in a straightforward manner. Algorithm GRD calls RSR_1, RSR_2, $\cdots$, $RSR_{\lfloor \frac{n}{2} \rfloor}$ and outputs the shortest schedule among the schedules generated by RSR_k's.

In the case of a PE failure, however, schedules generated by GRD often become much longer than no failure case. The reason is explained briefly as follows: At Step 2 of RSR_k, instances can be scheduled only to locations that are not occupied by the instances scheduled at Step 1. As a result, the instances scheduled at Step 2 tend to be located on later time slots compared to instances scheduled at Step 1. Thus the instances scheduled at Step 2 can incur delay of the finish time of the program.

In order to overcome this problem, we introduced the following idea: the set of all tasks is partitioned into some disjoint small subsets, and then Step 1 and Step 2 are repeatedly executed for each subset. Then, for each task, its instances scheduled at Step 1 and those duplicated at Step 2 become closer to each other in the resultant schedule than GRD, and the degree of degradation in performance in the case of a PE failure can be decreased. Based on this idea, we have developed Algorithm PHS^l. In PHS^l, all tasks are ordered according to their levels, and partitioned *equally* into l disjoint subsets. Then Algorithm GRD is applied to each subset.

Example 2 Figure 3 illustrates how Algorithm PHS^3 generates a fault-tolerant schedule. Suppose that the number of PEs is four. Given a task graph in Figure 3(a), the set of tasks is partitioned equally into three subsets as follows:

$$G_1 = \{v_1, v_2, v_3, v_4, v_6\}, G_2 = \{v_5, v_7, v_8, v_9, v_{10}\},$$
$$G_3 = \{v_{11}, v_{12}, v_{13}, v_{14}\}$$

(The number of tasks in G_3 is four because that of all tasks in the task graph is 14, which is indivisible by three.)

First, Algorithm GRD is applied to G_1. GRD calls RSR_1 and RSR_2. Using $n-1$ PEs, i.e., p_1, p_2 and p_3, RSR_1 generates a subset of schedule (called a partial schedule) shown in Figure 3(b) at Step 1. Then at Step 2, tasks that are not duplicated at Step 1 (i.e., v_2, v_3, v_4 and v_6) are scheduled using n PEs as shown in Figure 3(c). (v_1 on p_4 is an instance duplicated for improving the start time of the instance of v_2 on p_4.) Similarly, RSR_2 is applied to G_1. In this case, the partial schedule generated by RSR_1 is chosen. GRD is iteratively applied to the remaining subsets G_2 and G_3. As a result, a fault-tolerant schedule is obtained as shown in Figure 3(d).

In general, the length of the obtained schedule critically depends on the structure of the given task graph. However, Algorithm PHS^l partitions a set of tasks without enough consideration of the structure. Additionally, determining an appropriate value of l is also a problem since the best value that minimizes a resultant schedule length is different for different task graphs. Another drawback is that the running time is relatively large. This is because in PHS^l, all of Algorithms $RSR_1, RSR_2, \cdots, RSR_{\lfloor \frac{n}{2} \rfloor}$ are applied to each partitioned subset.

4 The Proposed Scheduling Algorithm

In this section, we present the proposed scheduling algorithm. To cope with the shortcomings of the previous algorithm, the proposed algorithm employs a new partitioning method and a simple scheduling scheme. The outline of this algorithm is as follows:

Proposed Algorithm
Input: TG, a task graph;
P, a set of PEs $\{p_1, p_2, \cdots, p_n\}$ $(n \geq 2)$
Output: S, a fault-tolerant schedule
Begin
S := empty
Partitioning:
Partition a set of tasks in TG into task groups $G_1, G_2, \cdots, G_m$ according to height.
{Task groups are arranged in descending order of height.}
Applying Basic algorithm to each task group:
For $i = 1$ to m do
$S := BA(G_i, S)$
End_For
End

4.1 Partitioning

Introducing a new notion of *height* of a task, we propose to partition a set of tasks according to their heights. The height of a task v is defined as

$$height(v) = \begin{cases} 0, & \text{if } v \text{ has no immediate successors,} \\ 1 + \max_{u \in U}\{height(u)\}, & \text{otherwise,} \end{cases}$$

where U is a set of immediate successors of v.

Partitioning is performed as follows: Given a task graph, the height of each task is calculated first. Then, a set of tasks is partitioned into subsets according to their heights in such a way that all tasks with the same height will belong to one subset. We call each subset a *task group*. By definition, for any two tasks $v, v' \in V$, if v is a preceding task of v', then v has larger value of height than v'. Since all tasks in each task group have the same value of height, they have no data dependencies (precedence constraints) among them. As will be analyzed in Section 5, tasks can be scheduled effectively due to this property. In addition, since the height of a task is uniquely determined, task groups are determined uniquely too.

Example 3 Consider a task graph in Figure 1(b). For example, the heights of v_4 and v_7 are 2 and 1, respectively. The set of all the tasks is partitioned into five task groups as follows:

$$G_1 = \{v_1\}, G_2 = \{v_2, v_3\}, G_3 = \{v_4, v_5, v_6\}, G_4 = \{v_7, v_8\}, G_5 = \{v_9\}$$

4.2 Basic Algorithm

Once the program has been partitioned into task groups, Basic algorithm described in this section is applied to each task group. This algorithm consists of two steps.

At Step 1, each task is scheduled to one of $n-1$ PEs, i.e., p_1, p_2, $\cdots$, p_{n-1}. The tasks are scheduled one by one according to their priorities (the task at the highest priority is scheduled first). Priorities are assigned in descending order of level. Tasks at the same level are prioritized according to the number of their immediate successors (the task with the greatest number of immediate successors is prioritized highest).

Now let $v \in G_i$ be the task to be scheduled. Note that all tasks in G_1, G_2, $\cdots$, G_{i-1} are already scheduled, i.e., a partial schedule S' already exists. Then, v is scheduled to one of the $n-1$ PEs by adding its instance to S'. The location of v is determined as follows: For each PE, the earliest start time of v on the PE is computed, provided that on the PE, v is not executed earlier than any instance in S'. This can be done by calling Procedure TDP [7, 9] for each of the $n-1$ PEs. Then v is scheduled to the PE which can execute it the earliest of all the PEs.

At Step 2, all tasks in the task group are duplicated. The newly duplicated tasks are scheduled in the same order as Step 1. The location of each of them is determined in a similar way to Step 1, except that each instance is never scheduled to the PE where its corresponding task is already scheduled at Step 1. Consequently, every task is allocated to at least two different PEs. The following is the pseudo-code of Basic algorithm.

Basic algorithm $BA(G_i, S')$
Input: G_i, a task group;
S', a partial schedule
Output: S, a partial schedule
Begin
Arrange tasks in G_i according to their priorities
Step 1:
For each task v in G_i do
For each PE p in $P - \{p_n\}$ do
$DTlst[p]$:= NULL {$DTlst$ is a list containing duplicated predecessors.}
$ST[p] := TDP(v, p, DTlst[p])$
{$ST[p]$ is the earliest start time of v on p.}
End_For
p_t := the PE whose $ST[p_t]$ is the smallest
Schedule v with $DTlst[p_t]$ to p_t at time $ST[p_t]$
End_For
Step 2:
For each task v in G_i do
p_a := the PE to which v has been scheduled at Step 1
For each PE p in $P - \{p_a\}$ do
$DTlst[p]$:= NULL
$ST[p] := TDP(v, p, DTlst[p])$
End_For
p_t := the PE whose $ST[p_t]$ is the smallest
Schedule v with $DTlst[p_t]$ to p_t at time $ST[p_t]$
End_For
End

Due to the space limitation, we omit the proof of the correctness of the proposed algorithm.

The complexity of task level and height calculation is $O(|E|)$, where $|E|$ denotes the number of edges in the task graph. Each task is scheduled by applying

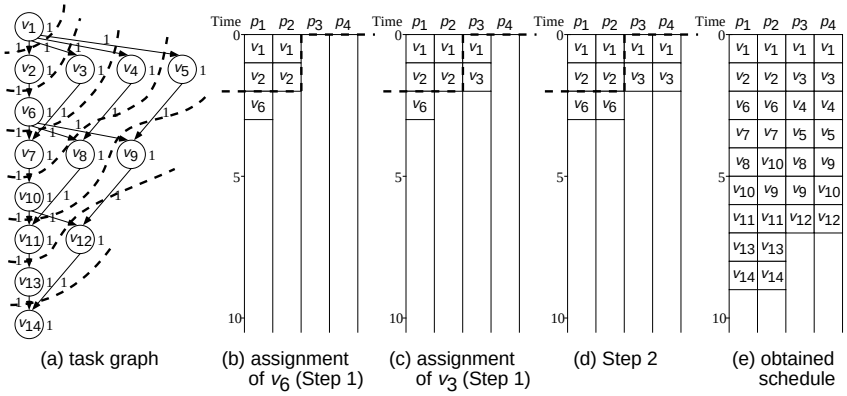


Figure 4: Illustrative example of the proposed algorithm.

Procedure TDP to $n-1$ PEs both at Step 1 and Step 2 of Basic algorithm. The computational complexity of Procedure TDP is known to be $O(|V|^3)$ [9], where $|V|$ denotes the number of tasks in the task graph. Therefore, the complexity of scheduling of one task is $O(n|V|^3)$. Since $|E| < |V|^2$ and the number of task is $|V|$, the complexity of the proposed algorithm is $O(|V|^4)$, given that n is fixed.

4.3 Illustrative Example

Figure 4 illustrates how the proposed algorithm works. In this example, the number of PEs n is assumed to be four, and a task graph shown in Figure 4(a) is given. A set of tasks is partitioned into eight task groups $G_1, G_2, \cdots, G_8$. Tasks in each task group are ordered according to their priorities as follows:

G_1:	v_1	G_5:	v_{10}, v_5, v_8
G_2:	v_2	G_6:	v_9, v_{11}
G_3:	v_6, v_3	G_7:	v_{12}, v_{13}
G_4:	v_4, v_7	G_8:	v_{14}

These task groups are ordered according to their heights. Then Basic algorithm is applied for each task group in the order. The task group whose height is the largest is selected first. Note that Step 1 and 2 are applied only once to each task group, unlike in the previous algorithm.

Now suppose that task groups G_1 and G_2 have been scheduled. Then Basic algorithm is applied to G_3. At Step 1, each task in G_3 is scheduled to one of $n-1$ PEs, i.e., p_1, p_2 and p_3. This is done by applying Procedure TDP to each of the three PEs. For example, instance of v_3 is scheduled as follows: The instance of v_6 has been already assigned to p_2 as shown in Figure 4(b). Taking account of the immediate predecessor of v_3 (i.e., v_1), it is seen that the start times of v_3 on p_1 and p_2 are 3 and 2 respectively. The start time of v_3 is also 2 on p_3 without duplication of task. (Note that v_3 must receive necessary data from v_1.) In order to improve the start time of v_3, TDP duplicates the instance of v_1 and schedules it to p_3 at time 0. By this duplication, the communication delay between v_1 and v_3 is eliminated and the start time of v_3 on p_3 becomes 1. As a result, it is found that p_3 can start execution of v_3 earlier than p_1 and p_2. Therefore, the instance of v_3 is scheduled to p_3 as shown in Figure 4(c).

At Step 2, each task in G_3 is duplicated and scheduled to one of all n PEs except the PE to which its instance is already scheduled. The instance of v_3 is already scheduled to p_3 at Step 1, then TDP is applied to p_1, p_2 and p_4. As a result, an instance of v_3 is scheduled to p_4. Similarly, each remaining task is scheduled so as to be executed on two different PEs as shown in Figure 4(d).

Basic algorithm is applied to the remaining task groups $G_4, G_5, \cdots, G_8$. Consequently, a fault-tolerant schedule is obtained as shown in Figure 4(e).

5 Experimental Evaluation

We performed simulation studies using a large number of task graphs as workload. In this section, we present a performance comparison between the proposed algorithm and the previous algorithm. We also compare the running times of these algorithms on a Sun UltraSPARC UA1 workstation.

5.1 Simulation Environment

In the previous study, it is found that by partitioning a set of tasks into a few subsets, good performance can be obtained [7]. In this evaluation, we selected PHS^2, PHS^3 and PHS^5 as the previous algorithms for a fair comparison with the proposed algorithm.

In addition, as in [7], we introduce a straightforward fault-tolerant scheduling algorithm STR. Algorithm STR duplicates a schedule of $n/2$ PEs entirely. Note that the finish time of any schedule generated by STR does not change when one PE has failed. Then we compute the finish time of schedules generated by STR.

As a baseline, we used the finish time of a (non-fault-tolerant) schedule generated by DSH. All results presented in this section are normalized to this length. In the studies, the number of PEs n is assumed to be 10.

5.2 Workload

In the simulation studies, we used task graphs for three practical parallel computations: Gaussian elimination [10], Laplace equation solver [15], and LU-

Table 1: Running times (sec) of the proposed algorithm and the previous algorithms.

	Proposed algorithm	PHS^2	PHS^3	PHS^5	STR
Gaussian elimination	0.33	2.33	2.52	2.45	0.13
Laplace equation solver	0.50	4.84	6.19	5.32	0.20
LU-decomposition	0.46	2.93	3.48	3.48	0.14

decomposition [11]. These task graphs can be characterized by the size of the input matrix because the number of tasks and edges in the task graph depends on its size. For example, the task graph for Gaussian elimination shown in Figure 1(a) is for a matrix of size 3. The number of nodes in these task graphs is roughly $O(N^2)$ where N is the size of matrix. In the simulation, we varied the matrix sizes so that the graph size ranged from about 50 to 300 nodes for each kinds of parallel computation. For each size of the task graph, we generated seven different graphs for *ccr* equal to 0.1, 0.2, 0.5, 1.0, 2.0, 5.0 and 10.0 by varying communication delays. The *communication-to-computation ratio* (*ccr*) is defined as follows [1, 10]:

$$ccr = \frac{\text{average communication delay between tasks}}{\text{average execution time of tasks}}$$

5.3 Evaluation Results

Figures 5, 6, and 7 show the simulation results for various matrix sizes with $ccr = 2.0$ for (a) no PE failure case and (b) the worst case. As for the schedule lengths in the case of no PE failure, the proposed algorithm has unfortunately worse performance than the previous algorithms, although it outperforms STR for all sizes of matrix.

Next, we discuss the worst finish times of the schedules in the case of one PE failure. For the Gaussian elimination task graphs (Figure 5(b)), the proposed algorithm has better performance than the previous algorithms PHS^2, PHS^3 and PHS^5. Especially when the size of matrix is more than 13, the proposed algorithm exhibits better performance than Algorithm STR. For the Laplace equation solver task graphs (Figure 6(b)), the proposed algorithm outperforms the previous algorithms for almost all sizes of matrix, although STR shows better performance than the proposed algorithm in all sizes of matrix. For the LU-decomposition task graphs (Figure 7(b)), the proposed algorithm has better performance than STR for some sizes of matrix.

As explained in Section 3, the instances scheduled at Step 2 mainly cause the delay of the finish time of the program. By partitioning a set of tasks into some subsets, the delay of the finish time of schedule can be decreased, because for each task, the instance scheduled at Step 1 and that duplicated at Step 2 are scheduled closer in time to each other [7]. Subsets partitioned by the previous algorithm have no particular properties because they are partitioned equally. On the other hand, the proposed algorithm partitions a set of tasks according to height, which is one of the structural characteristics of a given task graph. By the definition of height, each task group has the property that all of the tasks have no data dependencies (precedence constraints) among them. From each task group, therefore, Basic algorithm can extract the maximum parallelism at both Step 1 and Step 2 (of Basic algorithm). As a result, the instance of each task duplicated at Step 2 is scheduled closer to its corresponding instance scheduled at Step 1 than the previous algorithm, and the degree of degradation of performance in the case of a PE failure can be decreased. Therefore, the proposed algorithm outperforms the previous algorithm in the case of one PE failure. Also, for the proposed algorithm, the degree of degradation of performance in no PE failure case is much smaller than that of improvement of the performance in the worst case as shown before compared with the previous algorithm. This is because the property described above can also contribute to the performance in no PE failure case. From these observations, we can conclude that the proposed algorithm improves the schedule length in the case of one PE failure at the cost of small degree of the degradation in the performance of no PE failure case.

Figures 8(a), 9(a), and 10(a) show the simulation results for the worst case with $ccr = 0.2$. (For space limitation, we omit the results for no PE failure case.) We can also see that the proposed algorithm outperforms the previous algorithms.

Figures 8(b), 9(b), and 10(b) show the simulation results for the worst case for the size of matrix equal to 16. We varied the value of *ccr* from 0.1 to 10.0 in this simulation. For all kinds of task graphs, as the value of *ccr* increases, the proposed algorithm shows better performance than STR. Also, it is seen that as the value of *ccr* increases, the fault-tolerance can be achieved with small decrease of performance compared with the non-fault-tolerant scheduling algorithm DSH (note that we used the schedule length of DSH as the baseline).

5.4 Comparison of Running Times

Finally, we compare the running time of the algorithms needed for scheduling a task graph. Table 1 shows the running times of the proposed algorithm, the previous algorithms PHS^2, PHS^3, PHS^5, and Algorithm STR. As inputs, the Gaussian elimination task graph with 250 nodes (size of matrix = 20), the Laplace equation solver task graph with 256 nodes (size of matrix = 16), and the LU-decomposition task graph with 252 nodes (size of matrix = 22) are used. The *ccr* is set to 2.0.

We can easily see that the running time of the proposed algorithm is much smaller than that of the previous algorithm although the time complexity of the previous algorithm is also $O(|V|^4)$ [7]. This is because the proposed algorithm has simpler structure than the previous algorithm. The reason why we can simplify

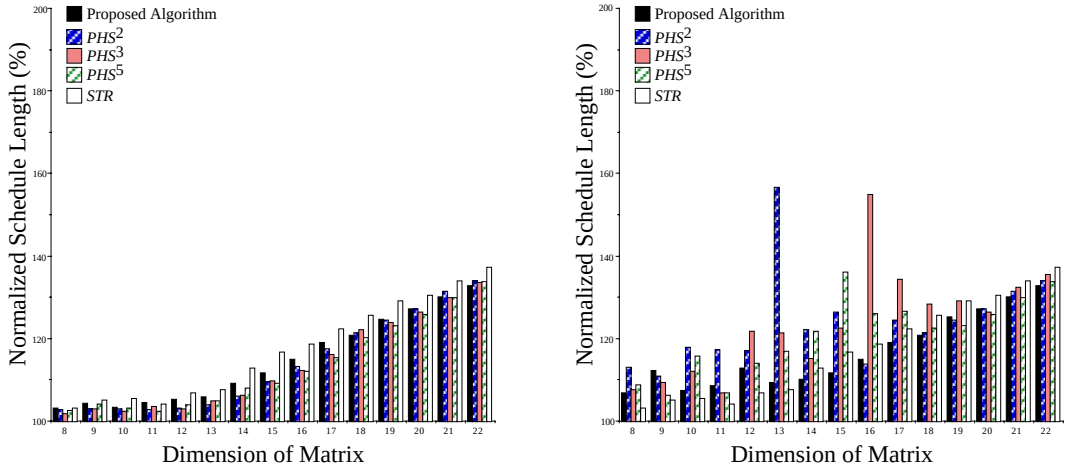


Figure 5: Results for Gaussian elimination task graphs with $ccr = 2.0$.

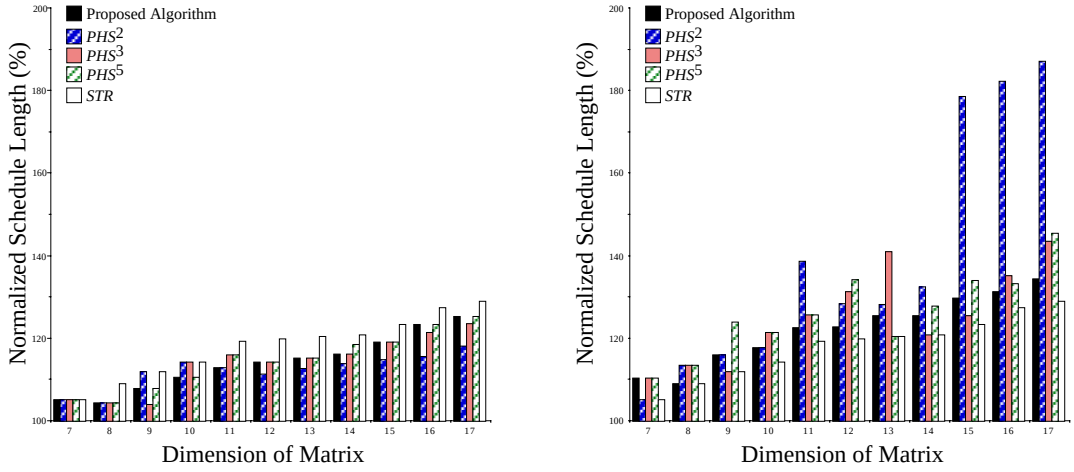


Figure 6: Results for Laplace equation solver task graphs with $ccr = 2.0$.

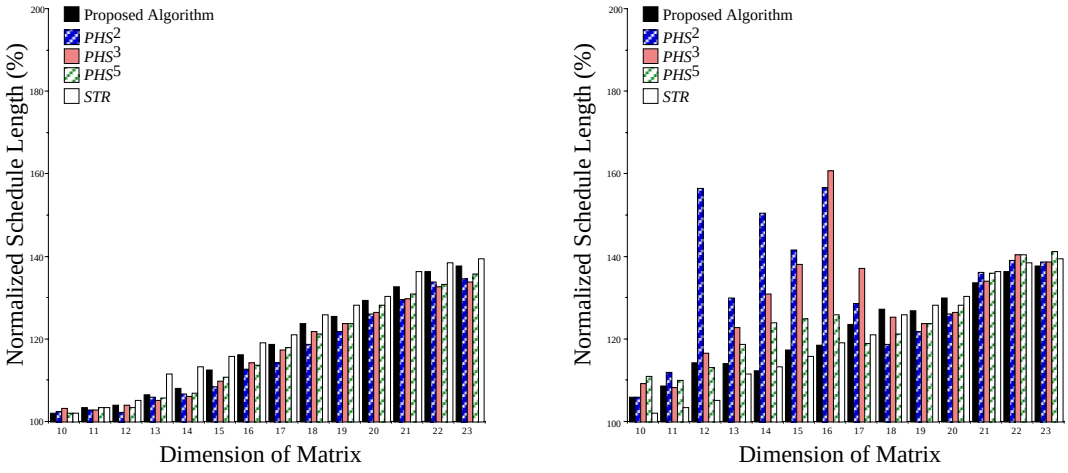
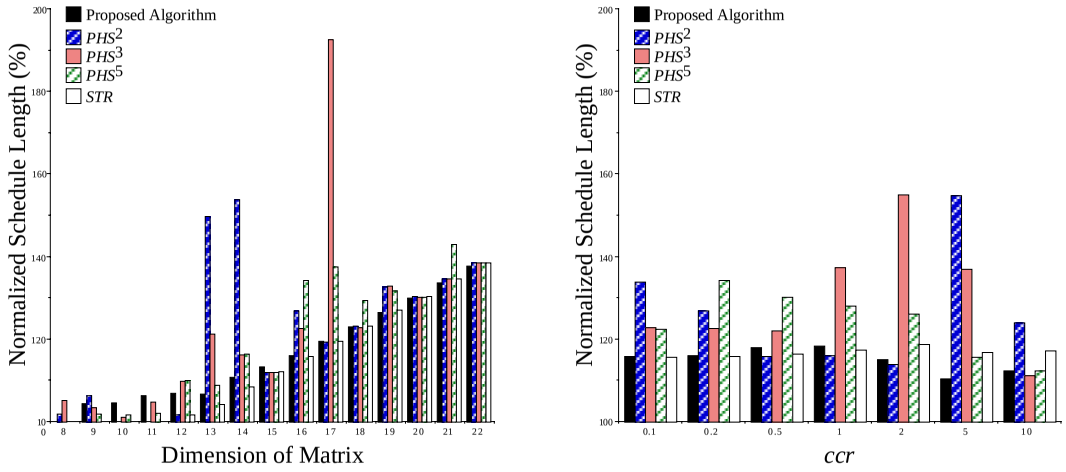
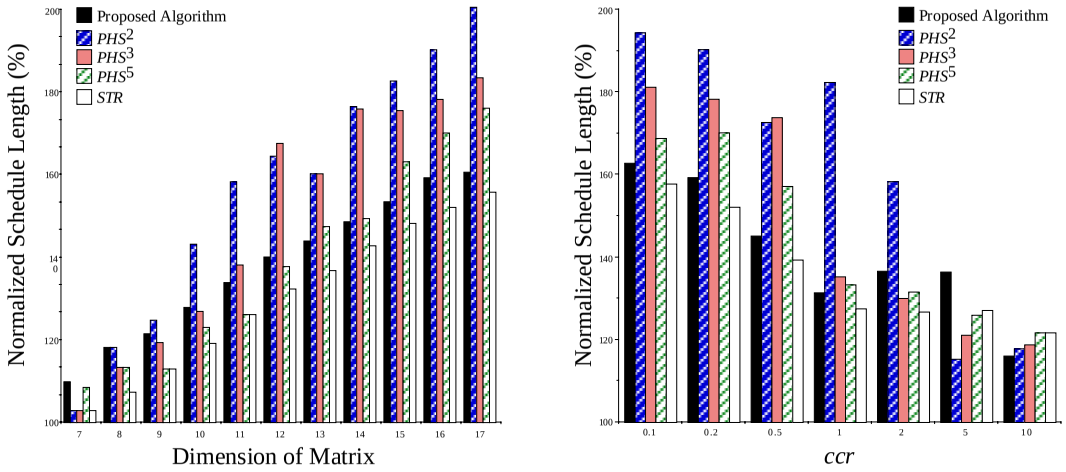


Figure 7: Results for LU-decomposition task graphs with $ccr = 2.0$.



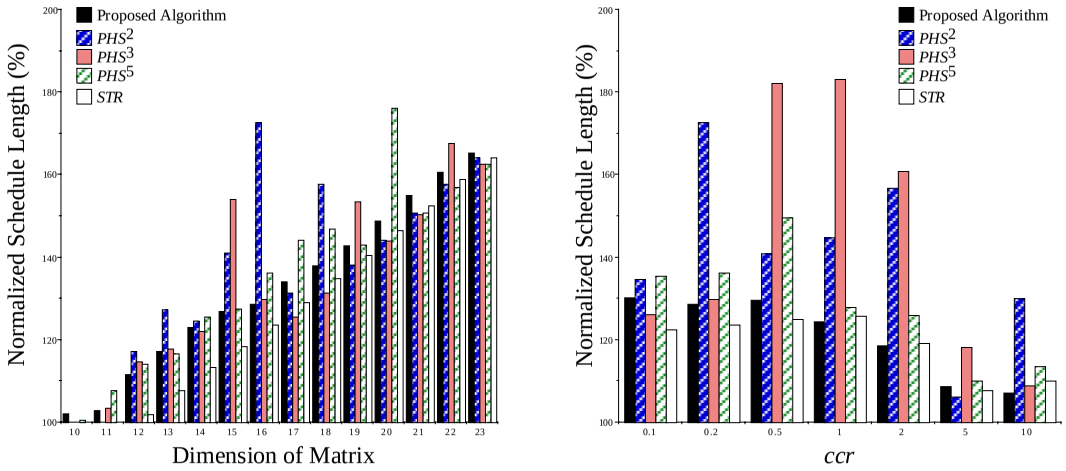
(a) *ccr* = 0.2 (b) matrix = 16

Figure 8: Results for Gaussian elimination task graphs in the worst case.



(a) *ccr* = 0.2 (b) matrix = 16

Figure 9: Results for Laplace equation solver task graphs in the worst case.



(a) *ccr* = 0.2 (b) matrix = 16

Figure 10: Results for LU-decomposition task graphs in the worst case.

the proposed algorithm is as follows: As shown in Section 3, the previous algorithm applies all of Algorithms RSR_1, RSR_2, $\cdots$, $RSR_{\lfloor \frac{n}{2} \rfloor}$ to each partitioned subset for achieving good performance, since the best value of k that minimizes the length of a resultant partial schedule is different for different subsets. This is because tasks in a partitioned subset have usually have precedence constraints mutually. On the other hand, the proposed algorithm applies Basic algorithm to each task group only once. As described before, all the tasks in each task group has no data dependencies among them. Therefore, using $n-1$ PEs at Step 1 is the best for obtaining short schedules.

The running time of STR is small since practically it has to consider only half of the PEs.

6 Conclusions

In this paper, we have proposed a new fault-tolerant scheduling algorithm for tolerating a single PE failure in multiprocessor systems. The time complexity of the proposed algorithm is $O(|V|^4)$ where $|V|$ is the number of tasks in the task graph.

In the proposed algorithm, a set of all tasks is partitioned into task groups according to their heights for achieving good performance. Since all tasks have no data dependencies in any task group generated by this partitioning, we were able to simplify the structure of the proposed algorithm.

We performed simulation studies using three kinds of practical parallel computation, i.e., Gaussian elimination, Laplace equation solver and LU-decomposition. The simulation results show that the proposed algorithm outperforms the previous algorithm particularly in the case of one PE failure. Moreover, the results show that the running time of the proposed algorithm is much smaller than that of the previous algorithm.

Currently we are implementing the proposed scheduling algorithm for a PVM-based network of workstations [4].

References

[1] I.Ahmad and Y.-K.Kwok, "A new approach to scheduling parallel programs using task duplication," Proc. of *International Conference on Parallel Processing*, pp.II-47-51, 1994.

[2] S.Chabridon and E.Gelenbe, "Failure detection algorithms for a reliable execution of parallel programs," Proc. of *14th International Symposium on Reliable Distributed Systems*, pp.229-238, 1995.

[3] V.Cherkassky and C.-I.H.Chen, "Redundant task-allocation in multicomputer systems," *IEEE Trans. Reliability*, vol.41, no.3, pp.336-342, Sep. 1992.

[4] A.Geist, A.Beguelin, J.Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, "PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing," Cambridge, Mass.: MIT Press, 1994.

[5] C.Gong, R.Melhem, and R.Gupta, "Loop transformations for fault detection in regular loops on massively parallel systems," *IEEE Trans. Parallel and Distributed Systems*, vol.7, no.12, pp.1238-1249, Dec. 1996.

[6] D.Gu, D.J.Rosenkrantz, and S.S.Ravi, "Construction and analysis of fault-secure multiprocessor schedules," Proc. of *21th International Symposium on Fault-Tolerant Computing*, pp.120-127, 1991.

[7] K.Hashimoto, T.Tsuchiya, and T.Kikuno, "A new approach to realizing fault-tolerant multiprocessor scheduling by exploiting implicit redundancy," Proc. of *27th International Symposium on Fault-Tolerant Computing*, pp.174-183, 1997.

[8] S.Kartik and C.Siva Ram Murthy, "Task allocation algorithms for maximizing reliability of distributed computing systems," *IEEE Trans. Computers*, vol.46, no.6, pp.719-724, June 1997.

[9] B.Kruatrachue, "Static task scheduling and grain packing in parallel processing systems," PhD dissertation, Electrical and Computer Eng. Dept., Oregon State Univ., Corvallis, 1987.

[10] Y.-K.Kwok and I.Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol.7, no.5, pp.506-521, May 1996.

[11] R.E.Lord, J.S.Kowalik, and S.P.Kumar, "Solving linear algebraic equations on an MIMD computer," *J. ACM*, vol.30, no.1, pp.103-117, Jan. 1983.

[12] R.D.Schlichting and F.B.Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Computer Systems*, no.1, vol.3, pp.222-238, March 1983.

[13] K.R.Pattipati, T.Kurien, R.-T.Lee, and P.B.Luh, "On mapping a tracking algorithm onto parallel processors," *IEEE Trans. Aerospace and Electronic Systems*, vol.26, no.5, pp.774-791, Sep. 1990.

[14] S.Tridandapani, A.K.Somani, and U.R.Sandadi, "Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault-detection and location," *IEEE Trans. Computers*, vol.44, no.7, pp.865-877, July 1995.

[15] M.Y.Wu and D.D.Gajski, "Hypertool: a programming aid for message passing systems," *IEEE Trans. Parallel and Distributed Systems*, vol.1, no.3, pp.330-343, July 1990.