PAPER

Effective Scheduling of Duplicated Tasks for Fault Tolerance in Multiprocessor Systems

Koji HASHIMOTO[†], **Tatsuhiro TSUCHIYA**[††], *and* **Tohru KIKUNO**[††], ***Regular Members***

SUMMARY In this paper, we propose a new scheduling algorithm to achieve fault tolerance in multiprocessor systems. This algorithm first partitions a parallel program into subsets of tasks, based on the notion of height of a task graph. For each subset, the algorithm then duplicates and schedules the tasks in the subset successively. We prove that schedules obtained by the proposed algorithm can tolerate a single processor failure and show that the computational complexity of the algorithm is $O(|V|^4)$ where V is the set of nodes of a task graph. We conduct simulations by applying the algorithm to two kinds of practical task graphs (Gaussian elimination and LU-decomposition). The results of this experiment 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.

key words: *multiprocessors, fault-tolerant scheduling, task graph, heights, task groups*

1. Introduction

Making use of multiple components for fault tolerance is a common idea in parallel and distributed computing. Many researchers have developed multiprocessor scheduling algorithms to achieve high reliability, assuming various system models.

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 a uniform execution time and communication delays between processors are not taken into account. In [6], [17], 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 only a single fault occurs.

Manuscript received March 26, 2001.

[†]The author is with Hitachi Laboratory, Hitachi Ltd., Hitachi-shi, 319–1292 Japan.

[††]The authors are with the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560–8531 Japan.

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 the method they propose is restricted to program structures which include only regular loops. A recent survey of such related work can be found in another paper of ours [9].

In this paper, we propose a new scheduling algorithm to tolerate a single processor failure in multiprocessor systems with a distributed memory architecture where 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. By duplicating every task of a given program, the algorithm ensures that the system can complete the program without rescheduling even if a single processor failure occurs. With this scheduling algorithm, fault tolerance can be achieved at the cost of small degree of time redundancy without requiring any additional hardware.

To the best of our knowledge, there has been no previous algorithm to the same problem discussed in this paper, except ours [7] (a part of [7] is also published with additional results as [8]). In the previous algorithm, a parallel program is partitioned equally into several subsets of tasks in the first phase, and then tasks in each subset are duplicated and scheduled successively. In this paper, we focus on the structure of task graphs and propose to use the structural information for task partitioning more directly than the previous algorithm. Specifically, we introduce a notion of *heights* [10] of tasks. The proposed fault-tolerant scheduling algorithm incorporates height-based task 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.

The remainder of this paper is organized as follows. The system model assumed in this paper is described in Sect. 2. Also, fault-tolerant schedules are defined in this section. The previous scheduling algorithm is explained in Sect. 3, while the new proposed algorithm is described in Sect. 4. The correctness proof of the proposed algorithm is given in Sect. 5. The results of the simulation studies are shown in Sect. 6. The paper concludes with Sect. 7.

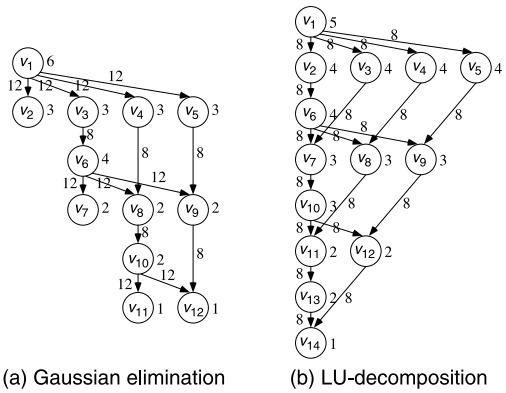


Fig. 1 Task graphs.

2. Preliminaries

2.1 System and Task Model

We consider a multiprocessor system that consists of n identical processing elements (PEs) and 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* [15].

A parallel program is 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', while v' is called an *immediate successor* of v. Similarly, if there is a path from v to v' on G, then v is called a *predecessor* of v', while v' is called a *successor* of v. Each edge is assigned a communication cost $c(v, v')$, which indicates the time required for transferring necessary 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*. Figure 1 shows examples of task graphs. In the figure, 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 [11] 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 representing the task to a node that has no successor nodes. In Fig. 1 (a), for example, the levels of v_6 and v_7 are 9 and 2, respectively.

2.2 Fault-Tolerant Scheduling

Multiprocessor scheduling refers to the process in which tasks in a task graph are assigned to PEs and the execution order of the tasks assigned to each PE is determined. Usually, the goal of scheduling is to minimize the length of the resultant schedules. When more than one copy is allowed to be scheduled for each task, that is, task duplication is allowed, a schedule can be formally defined as follows.

Definition 1 (Schedule): Let τ be a set of nonnegative real numbers. Given a task graph $TG = (V, E, w, c)$ and a set of PEs P, a *schedule* S is defined as a finite set of triples $S \subset V \times P \times \tau$, such that the following three conditions hold: (1) For each $v \in V$ there is at least one triple $(v, p, t) \in S$. (2) There are no two triples $(v, p, t), (v', p, t') \in S$ with $t' \leqq t < t' + w(v')$. (3) For each $(v, p, t) \in S$, if $v' \in V$ is an immediate predecessor of v, then there is another triple $(v', p, t') \in S$ with $t' < t$, or there is another triple $(v', p', t') \in S$ such that $p \neq p'$ and $t' + w(v') + c(v', v) \leqq t$. A triple (v, p, t) in S signifies that v is allocated to p and its starting time is t. A finite set of triples $S' \subset V \times P \times \tau$ is said to be a *partial schedule* iff $S' \subset S$ for some schedule S.

This definition is a natural extension of the one in [14]. In [14] (or in e.g., [1], [4], [11], [12], [16]), task duplication is used only to eliminate interprocessor communication to shorten the schedule length. Although the multiprocessor scheduling problem is known as NP-hard in its general form, when task duplication is allowed, the problem becomes easier and even tractable in some cases (e.g., [4]). On the other hand, fault-tolerant schedules must meet additional constraints, which will be described in the next subsection, thus making the scheduling problem more complicated.

Fault-tolerant scheduling, as discussed here, refers to producing a schedule with which the system can complete the program even if any single PE failure occurs. We call such a schedule *fault-tolerant*. The goal of our study is to minimize the schedule length while achieving this required level of fault tolerance. Before describing the formal definition of fault-tolerant schedules, we present an example to illustrate the basic concept.

Example 1: Figure 2 (b) shows a schedule for a task graph in Fig. 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 Fig. 2 (c). Even in this situation, all remaining instances that are not assigned to p_1 are executed in the same way as if the PE had not failed. On the other hand, if p_3 were to fail at time 0 as shown in Fig. 2 (d), the instances of v_{12} assigned

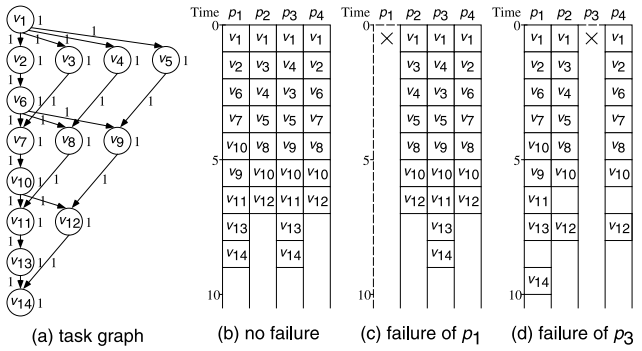


Fig. 2 An example of a fault-tolerant schedule.

to p_2 and p_4 could not be executed at their scheduled start time. This is because execution of the instance of its immediate predecessor v_9 scheduled on p_3 would not have 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 Fig. 2 (d).

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

2.3 Formal Definition of Fault-Tolerant Schedules

Suppose that a schedule S is given. Also suppose that $p^f (\in P)$ is a PE that has failed at time 0. Note that if the schedule can tolerate the failure of one PE at time 0, it can tolerate the failure of the same PE at any time $t > 0$. For each instance $(v, p, t) \in S$, let $start((v, p, t), p^f)$ and $finish((v, p, t), p^f)$ be, respectively, the (earliest) start time and the finish time of the instance when p^f has failed. Then, $finish((v, p, t), p^f)$ is equal to $start((v, p, t), p^f) + w(v)$. Let $ib(v, p, t)$ denote an instance that is scheduled immediately before (v, p, t) on the same PE p. If (v, p, t) is the instance that is scheduled earliest on p, then $ib(v, p, t)$ does not exist; i.e., $ib(v, p, t) = \varepsilon$. Let $start(\varepsilon, p^f) = finish(\varepsilon, p^f) = 0$.

As mentioned before, each instance (v, p, t) cannot start its execution until it receives all necessary data from its immediate predecessors. In addition, if $ib(v, p, t)$ cannot be executed, then (v, p, t) cannot be executed either. We assume that the start time of such an instance (v, p, t) is infinite, that is, $start((v, p, t), p^f) = \infty$. (v, p, t) is *executable* if $start((v, p, t), p^f) \neq \infty$, whereas (v, p, t) is *not executable* if $start((v, p, t), p^f) = \infty$.

For each instance (v, p, t) with $p \neq p^f$, the start time $start((v, p, t), p^f)$ is determined as follows.

Case 1: [v has no predecessors, or all immediate predecessors of v are scheduled on p earlier than t.]

$$start((v, p, t), p^f) = \max\{t, finish(ib(v, p, t), p^f)\}.$$

Case 2: [$ib(v, p, t)$ is not executable, or there is at least one immediate predecessor v' of v such that no instances of v' are executable.] In this case, (v, p, t) cannot start its execution, that is,

$$start((v, p, t), p^f) = \infty.$$

Case 3: [Otherwise] In this case, there is at least one immediate predecessor v' of v such that at least one of the instances of v' is executable and none of the executable instances is allocated to either p or p^f. The instance (v, p, t) has to wait to receive a message from such an immediate predecessor before starting. Hence,

$$\begin{aligned} &start((v, p, t), p^f) \\ &= \max\{t, finish(ib(v, p, t), p^f), \\ &\quad \max_{u \in U}\{\min_{\substack{\{(u,p',t')\} \\ |p' \neq p^f\}}}\{finish((u, p', t'), p^f) + c(u, v)\}\}\} \end{aligned}$$

where U is a set of all immediate predecessors of v whose instances are all allocated to PEs other than p.

Based on this argument, we obtain a formal definition of fault-tolerant schedules as follows.

Definition 2 (Fault-tolerant schedule): A schedule S is *fault-tolerant* iff the following two conditions hold for any $p^f (\in P)$.

(1) There is no triple $(v, p, t) \in S$ such that $p \neq p^f$ and $start((v, p, t), p^f) = \infty$.
(2) For each $v \in V$, there is at least one triple $(v, p, t) \in S$ with $p \neq p^f$.

The first condition signifies that the program can finish even if any single PE has failed, while the second condition stipulates that all tasks of the program can be executed.

3. Our Previous Work

As stated in Sect. 1, we have proposed a fault-tolerant scheduling algorithm in a previous paper [7]. In this section, we explain this previous algorithm.

In [7], we first proposed a set of scheduling algorithms, RSR_1, RSR_2, RSR_3, $\cdots$, as building blocks. Each RSR_k is an extension of Algorithm DSH (Duplication Scheduling Heuristic), which is a non-fault-tolerant scheduling algorithm proposed by Kruatrachue in [11]. 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

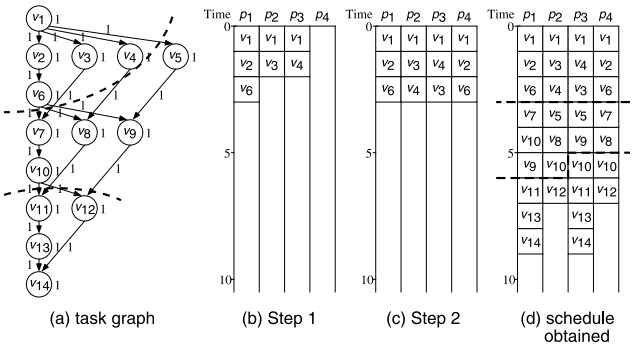


Fig. 3 Illustrative example of PHS^3.

2). The schedule obtained in Step 1 usually contains duplicated instances of tasks, because DSH duplicates tasks in order to eliminate communication delays and improve performance. In Step 2, tasks that were not duplicated in Step 1 are duplicated and scheduled.

In order to achieve 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 tend to be much longer than the no failure case. The reason can be described as follows. In Step 2 of RSR_k, instances can be scheduled only to locations that are not occupied by the instances scheduled in Step 1. Because of this, the instances scheduled in Step 2 tend to have later start times than the corresponding instances scheduled in Step 1. Thus instances scheduled in Step 2 can incur a delay in the completion of a program when a PE has failed. In order to avoid such a situation, we introduced the following idea.

In Algorithm RSR_k, Step 1 and Step 2 are executed only once for all tasks. Instead of this, we now consider partitioning the set of tasks V into l disjoint groups. Step 1 and Step 2 are then performed repeatedly for the tasks in each group. Consequently, the instances scheduled in Step 2 are likely to be distributed more equally among all the PEs, and also are likely to be scheduled into earlier time slots. Incorporating this idea, we developed Algorithm PHS^l. This algorithm *equally* partitions the set of tasks V into l disjoint subsets and schedules tasks in each subset by using Algorithm GRD.

Example 2: Consider the case of scheduling the task graph shown in Fig. 3 (a) onto four PEs by Algorithm PHS^3. Initially, the set of tasks V, in which all tasks are ordered according to their levels, is partitioned into three subsets: $G_1 = \{v_1, v_2, v_3, v_4, v_6\}$, $G_2 = \{v_5, v_7, v_8, v_9, v_{10}\}$, and $G_3 = \{v_{11}, v_{12}, v_{13}, v_{14}\}$. (The number of tasks in G_3 is four, since $|V|(= 14)$ is indivisible by three.)

First, tasks in G_1 are scheduled by Algorithm GRD. GRD calls RSR_1 and RSR_2. Using $n-1$ PEs, i.e., p_1, p_2 and p_3, RSR_1 generates a partial schedule as shown in Fig. 3 (b) in Step 1. In Step 2, tasks that are not duplicated in Step 1 (i.e., v_2, v_3, v_4 and v_6) are scheduled using n PEs as shown in Fig. 3 (c). (v_1 on p_4 is an instance duplicated to improve 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 Fig. 3 (d).

4. The Proposed Scheduling Algorithm

In general, the schedule length depends critically on the structure of the given task graph. However, Algorithm PHS^l partitions the set of tasks without sufficient consideration of its structure. Additionally, determining an appropriate value for l is a problem since the value that minimizes the 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 $RSV_1, RSV_2, \cdots, RSV_{\lfloor \frac{n}{2} \rfloor}$ are applied to each partitioned subset.

In this section, we present a new scheduling algorithm HBP (*Height-Based Partitioning*). To cope with the shortcomings of the previous algorithm, HBP employs a new partitioning method and a simple scheduling scheme. The outline of HBP is given below.

Proposed Algorithm HBP
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 the 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.*/
  Invoke Basic algorithm for each task group:
    For $i = 1$ to m do
      $S := BA(G_i, S)$
    End_For
End

4.1 Partitioning

We propose the partitioning of a set of tasks according to their *heights*. The *height* of a task v is defined as

$$height(v) = \begin{cases} 0, & U = \emptyset, \\ 1 + \max\limits_{u \in U}\{height(u)\}, & U \neq \emptyset, \end{cases}$$

where U is a set of immediate successors of v.

Partitioning is performed as follows. Given a task

graph, first the height of each task is calculated. Then, the 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 predecessor of v', then the height of v is larger than that of v'. Since all tasks in each task group have the same height value, there are no data dependencies (precedence constraints) among them. As will be analyzed in Sect. 3.5, this property allows tasks to be scheduled effectively. In addition, since the height of a task is determined uniquely, task groups are also determined uniquely.

Example 3: Consider the task graph in Fig. 1 (a). The heights of v_6 and v_7, for example, are 3 and 0, respectively. The set of all tasks is partitioned into eight task groups as follows. $G_1 = \{v_1\}$, $G_2 = \{v_2\}$, $G_3 = \{v_3, v_6\}$, $G_4 = \{v_4, v_7\}$, $G_5 = \{v_5, v_8, v_{10}\}$, $G_6 = \{v_9, v_{11}\}$, $G_7 = \{v_{12}, v_{13}\}$, and $G_8 = \{v_{14}\}$.

4.2 Basic Algorithm

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

In Step 1, each task is scheduled to one of n PEs. The tasks are scheduled one by one according to their priorities (the task with 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 immediate successors (the task with the greatest number of immediate successors is given the highest priority).

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} have already been scheduled; i.e., a partial schedule S' already exists. v is scheduled to one of the n 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 under the conditions that new instances of the predecessors of v are allowed to be scheduled onto the PE and no new instances (including the instance of v) are scheduled earlier than any instances that have already been scheduled on the PE in S'. This can be done by calling Procedure TDP (*Task Duplication Process*) [11] for each of the n PEs. When invoked, it returns the earliest start time of v and a set of instances of predecessors of v that need to be scheduled to achieve the start time. Finally v is scheduled to the PE which can execute v earliest among all the PEs.

In Step 2, all tasks in the task group are duplicated. The newly duplicated tasks are scheduled in the same order as in Step 1. The location of each task is determined in a similar way to Step 1, except that each instance is never scheduled to the same PE where the corresponding instance was already scheduled in Step 1. Consequently, every task is allocated to at least two different PEs. (Note that there may be tasks that are duplicated in the process of scheduling their successors, in order to eliminate communication delays.) The pseudo-code of the Basic algorithm is given below.

Basic algorithm $BA(G_i, S')$
Input: G_i, a task group;
S', a partial schedule
Output: S, a partial schedule
Begin
  $S := S'$
  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 do
        $(ST[p], DT[p]) := TDP(v, p, DT[p])$
        /*$ST[p]$ is the earliest start time of v on p.*/
        /*$DT[p]$ is a set of instances of
          duplicated predecessors of v.*/
      End_For
      $p_t :=$ the PE whose $ST[p_t]$ is the smallest
      $S := S \cup \{(v, p_t, ST[p_t])\} \cup DT[p_t]$
      /* Schedule v with $DT[p_t]$ to p_t */
    End_For
  Step 2:
    For each task v in G_i do
      $p_a :=$ the PE to which v has been scheduled
      in Step 1
      For each PE p in $P - \{p_a\}$ do
        $(ST[p], DT[p]) := TDP(v, p, DT[p])$
      End_For
      $p_t :=$ the PE whose $ST[p_t]$ is the smallest
      $S := S \cup \{(v, p_t, ST[p_t])\} \cup DT[p_t]$
    End_For
  Return S
End

4.3 Illustrative Example

Figure 4 illustrates how HBP works. In this example, we assume that the number of PEs, n, is four and the task graph shown in Fig. 4 (a) is given. The 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 first ordered according to their heights. Then the Basic algorithm is applied for each task group in order, such that the task group whose height is the largest is selected first. Note that Step 1 and Step 2 are applied only once to each task group, unlike in Algorithm PHS^l.

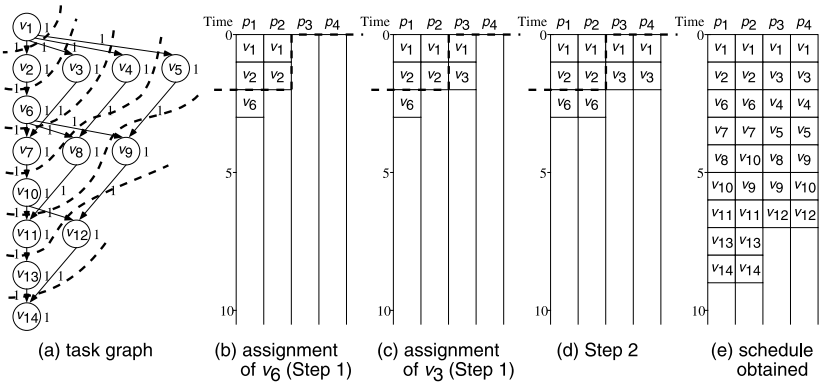


Fig. 4 Illustrative example of HBP.

Now suppose that task groups G_1 and G_2 have been scheduled. Then the Basic algorithm is applied to G_3. In Step 1, each task in G_3 is scheduled to one of n PEs. This is done by applying Procedure TDP to each of the three PEs. For example, an instance of v_3 is scheduled as follows. The instance of v_6 has already been assigned to p_2, as shown in Fig. 4 (b). Taking into account the immediate predecessor of v_3 (i.e., v_1), it can be seen that the start times of v_3 on p_1 and p_2 would be 3 and 2, respectively. The start time of v_3 would also be 2 on p_3 if no instances were duplicated. (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, 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 Fig. 4 (c).

In Step 2, each task in G_3 is duplicated and scheduled to one of the n PEs other than the PE to which its instance is already scheduled. The instance of v_3 is already scheduled to p_3 in Step 1, and thus 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 Fig. 4 (d).

The 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 Fig. 4 (e).

4.4 Time Complexity of Algorithm HBP

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 Procedure TDP to n PEs both in Step 1 and in Step 2 of the Basic algorithm. The computational complexity of Procedure TDP is known to be $O(|V|^3)$. Therefore, the complexity of scheduling one task is $O(n|V|^3)$. Since $|E| < |V|^2$ and the number of tasks is $|V|$, the complexity of Algorithm HBP is $O(|V|^4)$, given that n is fixed. This complexity is similar to many of non-fault-tolerant scheduling algorithms that use task duplication. For example, the complexity of DSH [11], $BTDH$ (*Bottom-Up Top-Down Duplication Heuristic*) [3], $LCTD$ (*Linear Clustering with Task Duplication*) [16], and the algorithm in [14] is $O(|V|^4)$, $O(|V|^4)$, $O(|V|^3 \log |V|)$, and $O(|V|^2(|E|+|V| \log |V|))$, respectively.

5. Correctness Proof of HBP

This section proves the correctness of Algorithm HBP. Correctness in this section means that the algorithm generates a fault-tolerant schedule. The correctness of HBP depends on the property of TDP that it never schedules instances earlier than those already scheduled. Although it is clear that the correctness of HBP depends on the correctness of TDP, we do not include the proof for TDP, since it is out of scope of our paper. Interested readers are referred to [11].

Without loss of generality, we discuss the case where a certain PE $p^f \in P$ failed at time 0 as in Sect. 2.3. Let S denote a schedule generated by HBP, and let $S_1, S_2, \cdots, S_m$ denote sets of the instances scheduled by applying the Basic algorithm to task groups $G_1, G_2, \cdots, G_m$, respectively (i.e., $S = S_1 \cup S_2 \cup \cdots \cup S_m$).

Lemma 1: For each task in $G_i (1 \leqq i \leqq m)$, there exists at least one instance in S_i that is not assigned to p^f.

Proof: Immediate from the fact that HBP assigns each task to at least two different PEs. □

Lemma 2: There is no instance $(v, p, t) \in S$ such that $p \neq p^f$ and $start((v, p, t), p^f) = \infty$. (That is, all instances on all PEs except p^f are executable even if p^f fails.)

Proof: We prove this by induction.

Base Step: By the definition of height, all tasks in G_1 have no predecessors. Thus S_1 contains only

instances of tasks in G_1 and they can start their execution without receiving any data from other instances. Due to the above property of TDP, for every instance $(v, p, t) \in S_1$, other instances that are not in S_1 are never scheduled earlier than the instance on p. Hence, every instance in S_1 is executable even in the case of the failure of p^f.

Induction Step: Assume that all instances in $S_1 \cup S_2 \cup \cdots \cup S_k$ are executable in the case of the failure of p^f. By Lemma 1, at least one instance of each task in $G_i (1 \leqq i \leqq k)$ that is not on p^f exists in S_i, and the assumption implies that all tasks that have larger values of height than any task in G_{k+1} have at least one executable instance in $S_1 \cup S_2 \cup \cdots \cup S_k$ that is not scheduled to p^f. In addition, on each PE, all instances in $S_1 \cup S_2 \cup \cdots \cup S_k$ have been scheduled earlier than any instance in S_{k+1}. Therefore, every instance in S_{k+1} can receive all necessary data from its immediate predecessors (which, it should be noted, all have larger values of height than the instance in question), unless it is not on p^f. This means that all instances in S_{k+1} that are not scheduled onto p^f are executable even in the case of the failure of p^f.

Consequently, all instances that are not on p^f in the schedule are executable; i.e., there is no instance $(v, p, t) \in S$ such that $p \neq p^f$ and $start((v, p, t), p^f) = \infty$. □

Lemma 1 implies that schedules obtained by HBP meet condition (2) in Definition 2 (in Sect. 2.3), whereas Lemma 2 implies that the schedules meet condition (1). Therefore, we obtain the following theorem.

Theorem 1: HBP generates a fault-tolerant schedule.

6. Experimental Evaluation

In this section, we present an experimental evaluation of the proposed algorithm. We performed simulation studies using a large number of task graphs as a workload. We coded the proposed algorithms in the C++ language and conducted the simulation on a COMPAQ XP1000 workstation running Digital UNIX.

6.1 Simulation Environment

In the simulation studies, we used task graphs for two practical parallel computations: Gaussian elimination [12] and LU-decomposition [13]. 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 the size. For example, the task graph for Gaussian elimination shown in Fig. 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 size so that the graph size ranged from about 100 to 400 nodes. For each task graph size, we generated six different graphs for *ccr* values of 0.1, 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], [12]:

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

As described before, the length of a fault-tolerant schedule may increase in the case of a PE failure. For the fault-tolerant schedule obtained, therefore, we calculate its length under two scenarios: the case of no PE failure, and the worst-case single PE failure. The schedule length in the worst case is calculated as follows. For each PE p^f, the schedule length in the case where p^f failed at time 0 is calculated. Then the largest value obtained is taken as the worst-case value.

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.

6.2 Comparison between HBP and PHS^l

In this study, we investigate the performance of HBP and PHS^l. For PHS^l, we selected 2, 3, and 5 as the values of l. Due to space limitations, we only present the results for the worst case scenario in this section. The omitted results show that HBP has almost the same performance as PHS^l in the case of no PE failure.

Figure 5 shows the results for (a) Gaussian elimination task graphs and (b) LU-decomposition task graphs with the value of $ccr = 5.0$ and the number of PEs = 16. As shown in this figure, HBP considerably improves the worst case performance.

As explained in Sect. 3, the delay of the completion time of program execution is mainly caused by instances scheduled in Step 2. The delay can be decreased by partitioning a set of tasks into small subsets, because this allows the instances scheduled in Step 2 to have similar start times to the corresponding instances scheduled in Step 1. Besides, HBP partitions a set of tasks according to their height, unlike PHS^l which simply partitions the set into l subsets of the same size. By the definition of height, each task group has the property that all its tasks have no data dependencies (precedence constraints) among them. Therefore HBP can exploit the maximum parallelism both in Step 1 and in Step 2. As a result, the instances scheduled in Step 2 tend to be assigned into earlier time slots than PHS^l, thus decreasing the degree of degradation in performance in the case of a PE failure.

6.3 Comparison between HBP and STR

In the previous subsection, we show that HBP outperforms our previous algorithm. Here we discuss the usefulness of our approach. To the best of our knowledge,

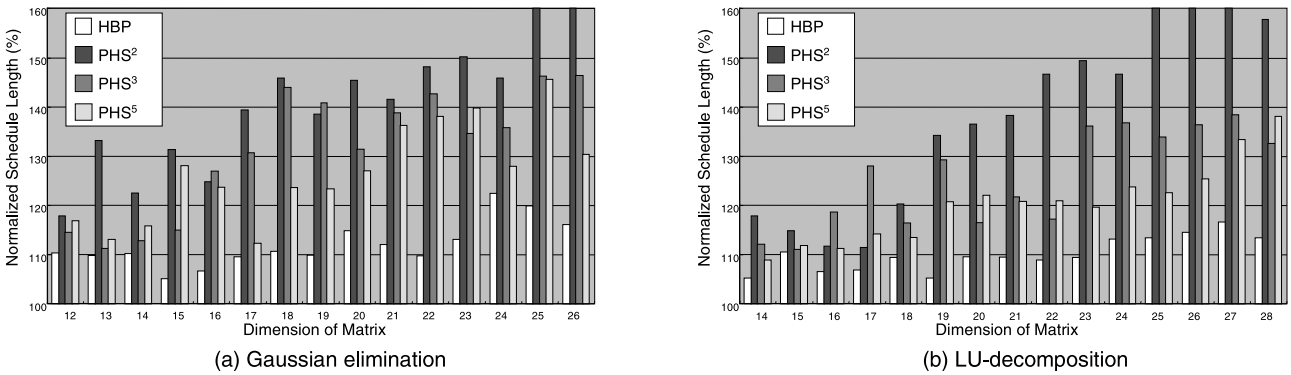


Fig. 5 Results for # of PEs = 16 and $ccr = 5.0$.

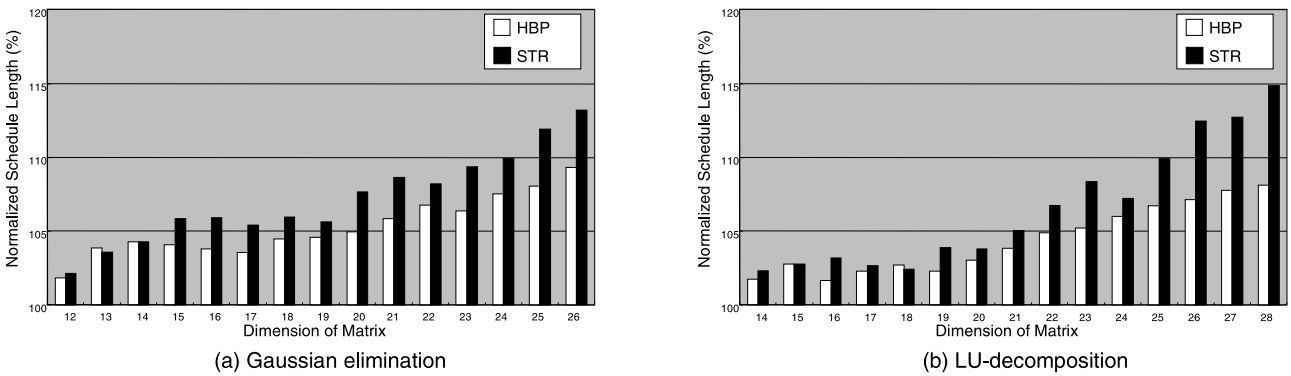


Fig. 6 Results for # of PEs = 16 and $ccr = 5.0$.

no fault-tolerant scheduling algorithms have been developed by other researchers that can deal with general task graphs with arbitrary computation and communication costs. To show the usefulness of HBP, therefore, we introduce a straightforward fault-tolerant scheduling algorithm STR and conduct a comparison between the two algorithms.

Algorithm STR
Input: TG, a task graph;
n, the number of PEs ($n \geq 2$)
Output: S, a fault-tolerant schedule
Begin
Generate a schedule S' for $\lfloor \frac{n}{2} \rfloor$ PEs by applying a non-fault-tolerant scheduling algorithm DSH [11].
Generate a fault-tolerant schedule S by duplicating S'.
End

The time complexity of STR is $O(|V|^4)$ since DSH is of complexity $O(|V|^4)$.

In the following, we present the simulation results for the case of no PE failure. These results show that HBP achieve substantial improvement over STR for a large range of parameter values. Unfortunately, omitted results show that the performance of HBP in the worst case is poorer than that of STR. However, we do not think of this as a serious disadvantage of HBP, because in practice, the gain in performance in no failure case is more significant than than in failure case. In addition, the difference in performance was within 10% on average,

Figure 6 shows the results for Gaussian elimination task graphs and LU-decomposition task graphs, when the value of ccr and the number of PEs are fixed to 5.0 and 16, respectively. The results show that HBP outperforms STR for almost every size of matrix. Furthermore, as the matrix size increases, the difference between the performance of HBP and that of STR increases. The following reason is conjectured. In general, the increase in task graph size also increases the maximum number of tasks that can be executed in parallel at a time. We can exploit such parallelism only if we have a sufficient number of PEs. In this simulation, the number of PEs n is fixed regardless of the size of task graph. Therefore, as the matrix size increases, HBP can take advantage of more parallelism than STR because STR can essentially use only $n/2$ PEs.

Figure 7 shows the results when the matrix size is 24 and the number of PEs is 16. In this simulation, we varied the value of ccr from from 0.1 to 10. In both kinds of task graphs, HBP exhibits better performance than STR for a large range of ccr values.

Finally, Fig. 8 shows the results obtained by vary-

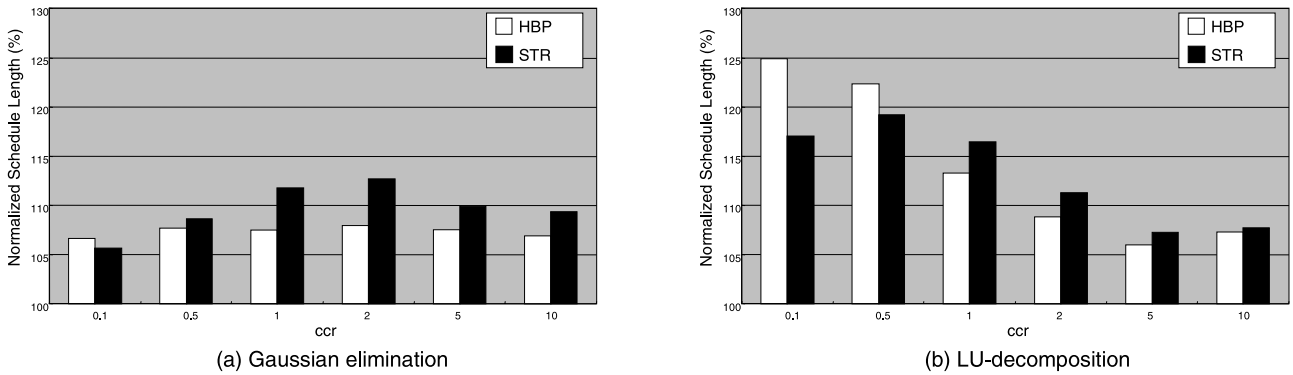


Fig. 7 Results for # of PEs = 16 and matrix size = 24.

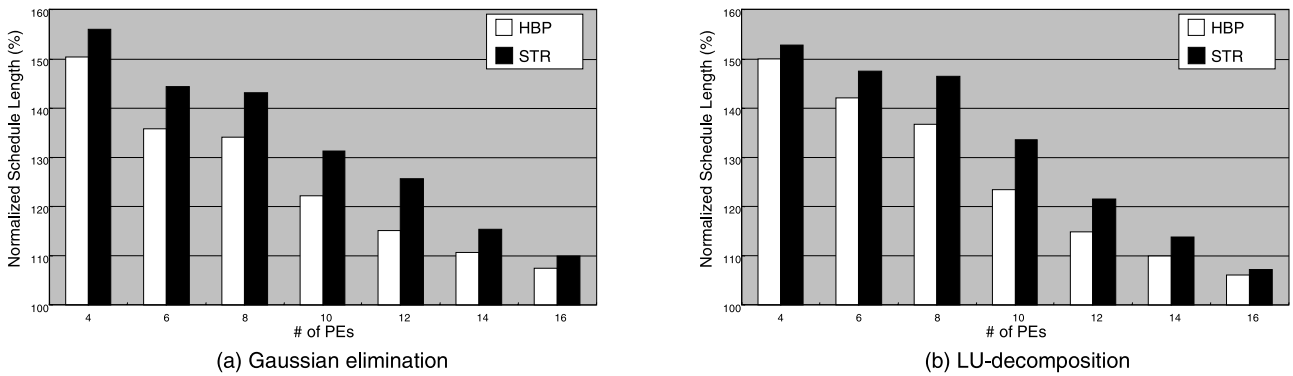


Fig. 8 Results for matrix size = 24 and $ccr = 5.0$.

Table 1 Running times (sec) of the algorithms.

Matrix sizes	HBP	PHS^2	PHS^3	PHS^5	STR
12	0.17	1.25	1.28	1.48	0.02
14	0.23	1.91	1.95	2.21	0.03
16	0.28	3.07	3.33	4.00	0.03
18	0.37	6.27	8.03	9.15	0.05
20	0.52	13.4	20.6	23.8	0.08
22	0.65	39.7	59.6	83.4	0.10
24	0.90	121	175	276	0.12
26	0.98	469	640	995	0.15

ing the number of PEs. The matrix size and the *ccr* value are fixed to 24 and 5.0, respectively. From the figure, it can be seen that for both HBP and STR, as the number of PEs is increased, the degradation in performance, which is inevitable in achieving fault tolerance, is reduced (note that the results are normalized to the length of non-fault-tolerant schedules generated by DSH.)

6.4 Comparison of Running Times

HBP has much better running times than PHS^l. Table 1 shows a comparison of the running times of the algorithms needed to schedule a task graph when the number of PEs is 16. The Gaussian elimination task graphs with the matrix size ranged from 12 (the number of tasks = 102) to 26 (the number of tasks = 403) were used as inputs.

In Table 1, one can easily see that the running time of HBP is much shorter than that of PHS^l, although the time complexity of PHS^l is also $O(|V|^4)$ [7]. This is due to the simplicity of HBP. As shown in Sect. 3, PHS^l invokes all RSR_1, RSR_2, $\cdots$, $RSR_{\lfloor \frac{n}{2} \rfloor}$ for every task group. On the other hand, HBP applies the Basic algorithm to each task group only once. The reason for the short running time of STR (which is also of complexity $O(|V|^4)$) is that it only needs to consider a half of the PEs.

7. Conclusions

In this paper, we addressed the problem of fault-tolerant scheduling for general task graphs. We proposed a new scheduling algorithm, which we refer to as HBP, to tolerate a single PE failure. We gave the correctness proof that any schedule generated by HBP is fault-tolerant. The time complexity of the proposed algorithm is $O(|V|^4)$, where $|V|$ is the number of tasks in the task graph.

HBP employs the partitioning of a set of tasks according to their heights. Each task group generated by this partitioning has the property that all its tasks have no mutual data dependencies. HBP can achieve high performance, because, due to this property, HBP can exploit the maximum parallelism to schedule tasks. This property also allows the scheduling algorithm itself to be simple, thus resulting in speedup in running

time. By conducting simulation studies, we empirically demonstrate these characteristics of the proposed algorithm.

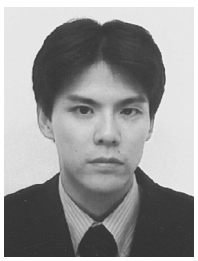
Acknowledgements

The authors are grateful to Mr. Aaron J. Stokes for a close reading of an earlier draft of this paper and many helpful suggestions on how to improve the presentation.

References

[1] I. Ahmad and Y.-K. Kwok, "A new approach to scheduling parallel programs using task duplication," Proc. International Conference on Parallel Processing, vol.2, pp.47–51, Aug. 1994.
[2] S. Chabridon and E. Gelenbe, "Failure detection algorithms for a reliable execution of parallel programs," Proc. 14th International Symposium on Reliable Distributed Systems, pp.229–238, Sept. 1995.
[3] Y.C. Chung and S. Ranka, "Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors," Proc. Supercomputing'92, pp.512–521, Nov. 1992.
[4] S. Darbha and D.P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," IEEE Trans. Parallel and Distributed Systems, vol.9, no.1, pp.87–95, Jan. 1998.
[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. 21th International Symposium on Fault-Tolerant Computing (FTCS-21), pp.120–127, June 1991.
[7] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "A new approach to realizing fault-tolerant multiprocessor scheduling by exploiting implicit redundancy," Proc. 27th International Symposium on Fault-Tolerant Computing (FTCS-27), pp.174–183, June 1997.
[8] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "A new approach to fault-tolerant scheduling using task duplication in multiprocessor systems," J. Systems & Software, vol.53, no.2, pp.159–171, 2000.
[9] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "Experimental evaluation of fault-secure scheduling under different error models in multiprocessor systems," IEICE Trans. Inf. & Syst., vol.E84-D, no.5, pp.635–650, May 2001
[10] T.C. Hu, "Parallel sequencing and assembly line problems," Operations Research, vol.9, no.6, pp.841–848, June 1961.
[11] B. Kruatrachue, Static task scheduling and grain packing in parallel processing systems, PhD Dissertation, Electrical and Computer Eng. Dept., Oregon State Univ., Corvallis, 1987.
[12] 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.
[13] 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.
[14] C.H. Papadimitriou and M. Yannakakis, "Toward an architecture-independent analysis of parallel algorithms," SIAM J. Computing, vol.19, no.2, pp.322–328, 1990.
[15] R.D. Schlichting and F.B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," ACM Trans. Computer Systems, vol.3, no.1, pp.222–238, March 1983.
[16] B. Shirazi, H. Chen, and J. Marquis, "Comparative study of task duplication static scheduling versus clustering and non-clustering techniques," Concurrency: Practice and Experience, vol.7, no.5, pp.138–153, June 1990.
[17] J. Wu, E.B. Fernandez, and D. Dai, "Optimal fault-secure scheduling," Comput. J., vol.41, no.4, pp.208–222, 1998.

Koji Hashimoto received the M.E. and Ph.D. degrees in computer engineering from Osaka University in 1997 and 2000, respectively. He is currently with Hitachi Laboratory of Hitachi Ltd. He is a member of IEEE.



Tatsuhiro Tsuchiya received the M.E. and Ph.D. degrees in computer engineering from Osaka University in 1995 and 1998, respectively. He is currently an assistant professor in the Department of Informatics and Mathematical Science at Osaka University. His research interests are in the areas of automatic verification and distributed computing. He is a member of IEEE.

Tohru Kikuno received M.E. and Ph.D. degrees from Osaka University in 1972 and 1975, respectively. He was with Hiroshima University from 1975 to 1987. Since 1990, he has been a professor in the Department of Informatics and Mathematical Science at Osaka University. His research interests include the quantitative evaluation of software development processes and the analysis and design of fault-tolerant systems. He served as a program co-chair of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98) and the Fifth International Conference on Real-Time Computing Systems and Applications (RTCSA '98). He also served as a general co-chair of the Second International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99).