Fault-Secure Scheduling of Arbitrary Task Graphs to Multiprocessor Systems

Koji Hashimoto
Hitachi Research Laboratory
Hitachi, Ltd.
Hitachi-city, Ibaraki 319-1292, Japan
khasimo@gm.hrl.hitachi.co.jp

Tatsuhiro Tsuchiya Tohru Kikuno
Department of Informatics and Mathematical Science
Osaka University
Toyonaka-city, Osaka 560-8531, Japan
{t-tutiya, kikuno}@ics.es.osaka-u.ac.jp

Abstract

In this paper, we propose new scheduling algorithms to achieve fault security in multiprocessor systems. We consider scheduling of parallel programs represented by directed acyclic graphs with arbitrary computation and communication costs. A schedule is said to be 1-fault-secure if the system either produces correct output for a parallel program or it detects the presence of any single fault in the system. Although several 1-fault-secure scheduling algorithms have been proposed so far, they can all only be applied to a class of tree-structured task graphs with a uniform computation cost. In contrast, the proposed algorithms can generate a 1-fault-secure schedule for any given task graph with arbitrary computation costs. Applying the new algorithms to two kinds of practical task graphs (Gaussian elimination and LU-decomposition), we conduct simulations. Experimental results show that the proposed algorithms achieves 1-fault security at the cost of small increase in schedule length.

1. Introduction

In recent years, much research has been conducted on methods for high reliability multiprocessor scheduling under various system models (e.g., [8, 13]). This paper focuses on fault-secure multiprocessor scheduling. The goal of fault-secure scheduling is to detect errors in computation of parallel programs carried out on multiprocessor systems. The basic approach to achieving fault security is to duplicate every task of a program and compare outputs of copies to ensure that either the output of the program is correct or at least one of the comparisons reports the existence of errors.

The concept of fault security was originally introduced in logic circuit design [7]. A circuit is *fault-secure* if for any single fault within the circuit, the circuit either produces correct output or produces a non-codeword. Banerjee and Abraham [1] first applied this concept to multiprocessor scheduling. Gu et al. [3] have investigated the formal characterization of fault-secure multiprocessor schedules by introducing the concept of *k-fault-secure scheduling*. In a k-fault-secure schedule, the output of a system is guaranteed to be either correct or tagged as incorrect for up to k processor faults. In their model, a parallel program is composed of a set of tasks and represented by a directed acyclic graph, and the number of processors is unlimited. Some scheduling algorithms have been proposed to achieve 1-fault security in [3]. More recently, Wu et al. [14] proposed an optimal fault-secure scheduling algorithm. Given the number of processors, the algorithm generates a 1-fault-secure schedule with the minimum schedule length.

However, the algorithms proposed in [3] and [14] assume that communication costs are negligible and all tasks have a uniform unit execution time. Moreover, these algorithms can only be applied to a class of tree-structured task graphs.

In this paper, we propose a new scheduling algorithm for achieving 1-fault security in multiprocessor systems with a distributed memory architecture, in which processors communicate with each other solely by message-passing. For comparison purposes, we also present a straightforward algorithm. We consider parallel programs represented by directed acyclic graphs with arbitrary computation and communication costs. Multiprocessor scheduling for most precedence-constrained task graphs is an NP-complete problem in its general form [2]. The algorithms we propose in this paper are heuristic; that is, schedules they produce are not necessarily optimal.

It is well known that inter-processor communication has serious effects on the performance of parallel processing. *Task duplication* [9] is an effective technique for improving the performance by reducing overheads of the communication. This technique eliminates communication delays by duplicating tasks among PEs. The technique thus improves the start times of tasks that need to wait for their preceding tasks, and also improves the finish time of the given program consequently.

In our approach to achieving 1-fault security, every task in the given task graph is replicated, and equality tests are carried out between the copies. The proposed algorithms schedule copies of tasks based on the task duplication technique to achieve better performance while maintaining 1-fault-secure properties.

2. Preliminaries

2.1. System and Task Model

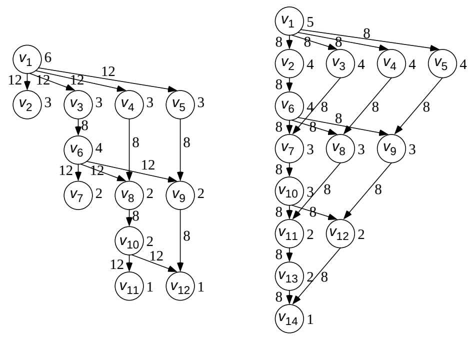
We consider a multiprocessor system that consists of n identical processing elements (PEs) and that runs one application program at a time. All PEs are fully connected with each other via a reliable network. A PE can execute tasks and communicate with another PE at the same time. This is typical with dedicated I/O processors and direct memory access.

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. If there exists a path from v' to v, v' is called a *predecessor* of v. A task that has no immediate successors is called an *output task*. 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 is zero. In the following, we call such a weighted DAG a *task graph*. Various applications are known to be represented by weighted DAGs (e.g., [12]). Figure 1 shows examples of task graphs. In the figure, the number adjacent to each node represents the execution time of the task represented by the node, and the number on each edge is the communication cost for data transfer.

We introduce some definitions and terminology as in [6]. 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 Figure 1(a), for example, the levels of v_6 and v_7 are 9 and 2, respectively. Finally, the *height* of a task is defined as

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

where U is a set of immediate successors of v. In Figure 1(a), for example, the heights of v_6 and v_7 are 3 and 0, respectively.



(a) Gaussian elimination (b) LU-decomposition

Figure 1. Task graphs.

2.2. Scheduling

In general, multiprocessor scheduling refers to the process in which tasks in a given task graph are assigned to PEs and the time slots in which the tasks are executed are determined. When more than one copy of each task is allowed to be scheduled, it is also necessary to specify from which copies to which copies data are transferred.

To distinguish between a task $v \in V$ and its scheduled copies, we call the latter the *instances* of v. We represent by $D(s) = (\alpha_1, \alpha_2, \cdots, \alpha_r)$ the fact that the instance s of v receives necessary data from $\alpha_1, \alpha_2, \cdots, \alpha_r$ which are instances of the immediate predecessors of v. By definition, r is equal to the number of the immediate predecessors. We always write α_i's in ascending order of the indices of the corresponding tasks.

Fault-secure scheduling, as discussed here, refers to producing a schedule with which even if any single fault occurs, the system can either produce the correct result for a given program or detect the fault. We call such a schedule a *1-fault-secure schedule* [3]. The goal of our research is to minimize the schedule length while achieving 1-fault security.

In our approach to achieving the 1-fault security property, every task $v \in V$ is replicated to produce at least two copies of its output and equality tests are carried out between different copies of some tasks. To do so, we need to allocate *tests* to PEs, in addition to the normal tasks of V. A test reports either "*equal*" or "*not equal*" according to the equality of the outputs of the copies compared. A fault is detected when some test reports "*not equal*". We assume that the outcome of a test carried out on a fault-free PE is always correct.

We use the notation $\tau(\alpha_1, \alpha_2, \cdots, \alpha_m)$ to indicate a test that compares the outputs of instances $\alpha_1, \alpha_2, \cdots, \alpha_m$ of the same task. Each test requires time for execution, and receiving data for comparison also incurs a communication delay. However the computation and communication costs of tests do not have any effect on the correctness of the proposed algorithms. Thus we do not introduce notations to represent these costs in this paper.

2.3. Fault Model and 1-Fault-Secure Schedule

We assume that a fault in a PE can result in errors in the outputs of an arbitrary set of instances of tasks and tests allocated to the PE. We call such a set a *fault pattern* if it is not empty, that is, a fault pattern is a non-empty subset of instances of tasks and tests that consists of all the instances whose outputs can be made erroneous directly by a fault in the system.

If a task receives erroneous outputs of other tasks, then the task itself may or may not become erroneous. We assume that an error in an instance of task causes a (possibly empty) subset of instances that receive data from that instance to be erroneous [4].

As in [3, 4], we introduce the notion of *interpretation*, which represents a possible error scenario.

Definition 1 (Interpretation) Given a schedule S, an *interpretation* I for S is a set $\sum = \{c, e, n, \mu_1, \mu_2, \cdots\}$ of *labels*, with distinguished labels "c", "e", "n", together with an assignment of a label to each instance of S such that:

1. each instance of a task is assigned a label from $\sum - \{e, n\}$, and
2. each instance of a test is labeled either "e" or "n".

In the definition, "c" means "*correct*", whereas μ_i represents an erroneous value of an output. Therefore, an instance of a task labeled "c' produces a correct output value, while an instance assigned a label μ_i outputs an erroneous value. The labels "e" and "n" represent the two possible outcomes of a test, "*equal*" and "*not equal*" respectively. In the following, we use $Label_I(s)$ as the label assigned by an interpretation I to an instance s of S.

The next definition gives the rule of producing scenarios for a given fault pattern.

Definition 2 (Consistency of Interpretation)
Given a schedule S and a fault pattern P', an interpretation I of S is *consistent* with P' if and only if the following conditions are satisfied.

(A) for an instance s of a task, if $Label_I(s) \neq$ "c", then either $s \in P'$ or there is at least one instance α in $D(s)$ such that $Label_I(\alpha) \neq$ "c".

(B) for an instance t of a test $\tau(\alpha_1, \alpha_2, \cdots, \alpha_m)$, if $Label_I(t) =$ "e", then either $t \in P'$ or $Label_I(\alpha_1) = Label_I(\alpha_2) = \cdots = Label_I(\alpha_m)$.

(C) for an instance t of a test $\tau(\alpha_1, \alpha_2, \cdots, \alpha_m)$, if $Label_I(t) =$ "n", then either $t \in P'$ or $Label_I(\alpha_i) \neq Label_I(\alpha_j)$ for some α_i and α_j $(i \neq j, 1 \leq i, j \leq m)$.

(D) for two instances, s and s', of a task, with $D(s) = (\alpha_1, \alpha_2, \cdots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \cdots, \alpha'_r)$, if $s, s' \notin P'$ and $Label_I(\alpha_q) = Label_I(\alpha'_q)$ for all q $(1 \leq q \leq r)$, then $Label_I(s) = Label_I(s')$.

(E) for two instances, s and s', of a task, with $D(s) = (\alpha_1, \alpha_2, \cdots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \cdots, \alpha'_r)$, if $s, s' \notin P'$ and there exists at least one α_q $(1 \leq q \leq r)$ such that $Label_I(\alpha_q) \neq Label_I(\alpha'_q)$, then $Label_I(s) \neq Label_I(s')$.

Condition (A) implies that in any valid scenario for a fault pattern P', the output of s can be erroneous only if either s is computed on the faulty PE or one of the instances of the immediate predecessors of v is erroneous. Conditions (B) and (C) indicate that the outcome of a test carried out on a non-faulty PE is determined by the labels of instances participating in the test, while a valid scenario may assign "e" or "n" arbitrarily to tests carried out on the faulty PE. Condition (D) states that different instances of a task computed on non-faulty PEs with identical input values must have the same output value. Condition (E) is the assumption that instances of a task computed on non-faulty PEs with different input values from each other must have different output values.

Based on the concept of interpretation, we define a 1-fault-secure schedule as follows.

Definition 3 (1-Fault-Secure schedule) A schedule S is 1-fault-secure if and only if for every fault pattern P' and for every interpretation I that is consistent with P', $Label_I(s) =$ "c" for every instance s of every output task, or there exists at least one instance t of a test such that $Label_I(t) =$ "n".

3. 1-Fault-Secure Scheduling Algorithms

In this section, we present two scheduling algorithms STR and $TDFS$ to achieve the 1-fault security property. Both algorithms tag each instance with "0" or "1", which we call the *version number*.

3.1. Straightforward Algorithm

A simple way of achieving 1-fault security is to simply duplicate a non-fault-secure schedule. We refer to this algorithm as STR. Algorithm STR produces the non-fault-secure schedule by applying DSH, which is a (non-fault-secure) scheduling algorithm proposed by Kruatrachue [9]. Figure 2 shows an example of applying STR. In the figure,

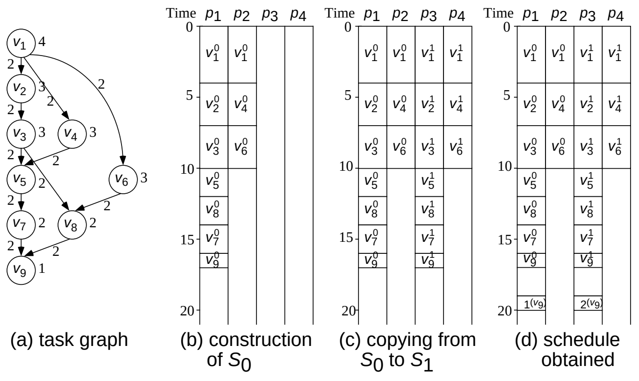


Figure 2. Illustrative example of Algorithm STR.

v^i denotes an instance of v tagged with "i", and $\tau(v)$ denotes a test in which all instances of v participate. S_0 and S_1 are non-fault-secure schedules generated by DSH.

In a schedule generated by STR, every instance tagged with "0" exchanges necessary data only with other instances within S_0, while every instance tagged with "1" does so with other instances within S_1. In other words, each instance tagged with "0" never receive any data from instances tagged with "1", and vice versa. Clearly, therefore, the system needs to compare only the results of the output tasks. The method of scheduling tests is explained in the following subsection because it is common to Algorithm $TDFS$.

The time complexity of DSH is known to be $O(|V|^4)$ [9], where $|V|$ denotes the number of tasks in the task graph. As explained in Section 3.3, the complexity of scheduling one test is $(O(|V|^2)$. Therefore, the complexity of STR is $O(|V|^4)$.

3.2. Task Duplication Based Algorithm

In this section, we propose a new 1-fault-secure scheduling algorithm, which we refer to as $TDFS$. Algorithm $TDFS$ schedules each task based on *task duplication* [9], which can improve performance. $TDFS$ also tags every instance with either "0" or "1", and allocates tests by using this information. Unlike STR, however, $TDFS$ allows tasks with different version numbers to be allocated to the same PE. In addition to output tasks, therefore, it may be necessary to test other tasks. $TDFS$ examines whether a test is needed or not when each instance is scheduled. Tests are scheduled after all the instances of tasks have been scheduled. The outline of $TDFS$ is described below.

```
Algorithm $TDFS$
Input:  $G$, a task graph; $P$, a set of PEs $\{p_1, p_2, \cdots, p_n\}$ ($n \geq 2$)
Output:  $S$, a 1-fault-secure schedule
Begin
  $S$ := empty; $TQ$ := empty
  /*$TQ$ is a set of tasks that need to be tested.*/
  Partitioning:
    Partition the set of tasks in $G$ into task groups $G_1, \cdots, G_m$
    according to height.
    /*Task groups are arranged in descending order of height.*/
  Apply Basic algorithm $BA$ to each task group:
    For $i$ = 1 to $m$ do
      $S := BA(G_i, TQ, S)$
    End_For
    Put all output tasks into $TQ$
  Schedule tests for tasks in $TQ$:
    $S := TST(TQ, S)$
End
```

3.2.1 Partitioning

In $TDFS$, a given set of tasks is first 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*. For example, consider the task graph in Figure 1(a). The set of all tasks is partitioned into five task groups as follows.

$$G_1: \ v_1 \quad G_2: \ v_3 \quad G_3: \ v_3, v_6$$
$$G_4: \ v_5, v_8 \quad G_5: \ v_9, v_{10} \quad G_6: \ v_2, v_7, v_{11}, v_{12}$$

3.2.2 Basic Algorithm

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

In Step 1, all tasks in the given task group are scheduled and tagged with "0". 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 suppose that $v \in G_i$ is 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. In Step 1, v is scheduled to one of the n PEs by adding its instance, say s. All instances scheduled in Step 1, including s, are tagged with "0", The PE on which s will be placed is determined by repeating the following process for every PE.

First, the earliest start time of s is computed, given that s is scheduled on the PE. This can be done by calling Procedure TDP [9]. Once the start time of s on that PE has been obtained, instances from which s receives data are determined. According to the concept of task duplication, TDP may duplicate predecessors of v in order to improve the start time of s. Therefore, for each immediate predecessor ip of v, there often exists more than one instance of ip that can send data to s so that s can receive data before the designated start time. For each ip, the algorithm checks whether or not an instance of ip exists that is tagged with “0” and can deliver data to s before the start time. If there is such an instance, it is chosen; otherwise, another instance of ip which is tagged with “1” is chosen.

If predecessors of v are duplicated by TDP, instances which provide data to those predecessor instances are also determined as described above. Scheduling the new instance may necessitate testing some other tasks. Based on the state of data exchanges between instances, the algorithm determines which tasks, if any, need to be tested. Such tasks are put into a queue TQ called a *test queue*. The details of how these tasks are determined are shown later.

After repeating this process for all PEs, the task is scheduled to the PE that can execute it earliest among all the PEs. If there is more than one such PE, then a PE is chosen such that the tasks needed to be tested is minimized.

In Step 2, all tasks in the task group are duplicated and tagged with “1”. The newly duplicated copies are scheduled in the same order as in Step 1. The PE to each is scheduled is determined in the same way as in Step 1, except that (1) an instance is never scheduled to the same PE where its corresponding task was scheduled in Step 1, and (2) instances tagged with “1” rather than “0” are chosen first as instances for receiving data. Consequently, every task is allocated to at least two different PEs.

The tasks to be tested are determined as follows. Suppose that an instance s of v is scheduled on a PE p. Let i be the version number of s ($i = 0, 1$). Then we test every predecessor a of v that satisfies one of the following two conditions.

1. a is an immediate predecessor of v and s receives data from an instance of a that is tagged with “$1 - i$”, or

2. a is a predecessor of v and an instance of a that is tagged with “$1 - i$” is already assigned to p.

The pseudo-code of the Basic algorithm is given below.

Basic algorithm BA

```
Input:  $G_i$, a task group; $TQ$, a test queue; $S'$, 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 $P$ do
        $DTlst[p]$ := NULL
        /*$DTlst$ is a list containing duplicated predecessors of $v$.*/
        $TTlst[p]$ := NULL
        /*$TTlst$ is a list containing tasks that need to be tested.*/
        $ST[p]$ := $TDP(v, p, DTlst[p])$
        /*$ST[p]$ is the earliest start time of $v$ on $p$.*/
        $TTlst[p]$ := $CKT(v, p, DTlst[p], S')$
        /*Put tasks that need to be tested into $TTlst[p]$.*/
      End_For
      $p_t$ := the PE whose $ST[p_t]$ is the smallest
      Schedule $v^0$ with $DTlst[p_t]$ to $p_t$ at time $ST[p_t]$
      Put tasks in $TTlst[p_t]$ into $TQ$.
    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
        $DTlst[p]$ := NULL; $TTlst[p]$ := NULL
        $ST[p]$ := $TDP(v, p, DTlst[p])$
        $TTlst[p]$ := $CKT(v, p, DTlst[p], S')$
      End_For
      $p_t$ := the PE where $ST[p_t]$ is the smallest
      Schedule $v^1$ with $DTlst[p_t]$ to $p_t$ at time $ST[p_t]$
      Put tasks in $TTlst[p_t]$ into $TQ$.
    End_For
End
```

Algorithm for checking whether test is needed CKT

```
Input:  $v_a$, an assigned task; $p_a$, assigned PE candidate;
        $DTlst[p_a]$, a list of tasks duplicated; $S'$, a partial schedule
Output:  $TTlst[p_a]$, list of tasks needed to be tested;
Begin
  For each instance $v$ in $DTlst[p_a] \cup \{v_a\}$ do
    For each immediate predecessor $\alpha$ of $v$ do
      flag := $NECESSARY$
      If $\alpha$ is in $DTlst[p_a]$ Then flag := $UNNECESSARY$
      Else
        For each instance $\alpha_i$ of $\alpha$ in $S'$ do
          If (the arrival time of data from $\alpha_i$ to $v$
                $\leq$ the start time of $v$ on $p_a$) and
             (the version number of $\alpha_i$ = that of $v$) Then
             flag := $UNNECESSARY$; Break
          End_If
        End_For
      End_If
      If (flag = $NECESSARY$) Then put $v$ into $TTlst[p_a]$
    End_For
    For each predecessor $x$ of $v$ do
      If ($x$ is on $p_a$) and (the version number of $x \neq$ that of $v$)
        Then put $x$ into $TTlst[p_a]$
    End_For
  End_For
End
```

3.2.3 Scheduling of Tests

After the instances of all tasks have been scheduled, tests for the tasks in TQ are scheduled. A test for a task v is assigned to a PE p where neither instances of v nor instances of its predecessors are assigned. If there is more than one

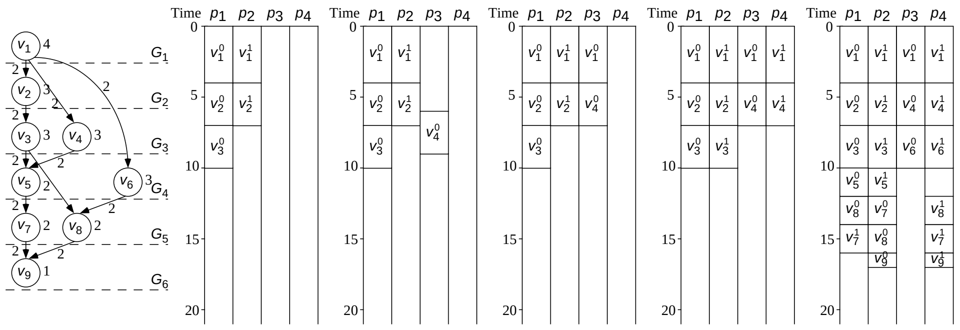

(a) task graph (b) assignment of v_5^0 (c) without duplication (d) duplication of v_1^0 (e) Step 2 (f) schedule obtained

Figure 3. Illustrative example of Basic algorithm.

qualifying PE, the one on which the test can be executed earliest is selected.

If there is no such PE, the test is duplicated and scheduled in such a way that each of the two copies is executed on a different PE from each other.

All the instances of v in S participate in the test. Note that there may be more than two instances of v in S, because v may be duplicated by Procedure TDP as successors of v are scheduled. Therefore, tests are not necessarily binary equality checks, unlike in [3, 4, 14].

Scheduling Algorithm for Tests TST

```
Input:  $TQ$, a test queue; $S'$, a partial schedule
Output:   $S$, a 1-fault-secure schedule
Begin
  For each task $v$ in $TQ$ do
    Find PEs to which no instances of $v$ or its predecessors are
    assigned, and put them into $AP$.
    If ($AP \neq$ NULL) Then
      For each PE $p$ in $AP$ do
        $ST[p]$ := the earliest start time of the test $\tau$ on $p$.
      $p_t$ := the PE where $ST[p_t]$ is the smallest
      Schedule $\tau$ to $p_t$ at time $ST[p_t]$
    Else
      For each PE $p$ in $P$ do
        $ST[p]$ := the earliest start time of the test $\tau_1$ on $p$.
      $p_{t_1}$ := the PE whose $ST[p_{t_1}]$ is the smallest
      Schedule $\tau_1$ to $p_{t_1}$ at time $ST[p_{t_1}]$
      For each PE $p$ in $P - \{p_{t_1}\}$ do
        $ST[p]$ := the earliest start time of the test $\tau_2$ on $p$.
      $p_{t_2}$ := the PE whose $ST[p_{t_2}]$ is the smallest
      Schedule $\tau_2$ to $p_{t_2}$ at time $ST[p_{t_2}]$
    End_If
  End_For
End
```

3.2.4 Time complexity

The complexity of task level and height calculation is $O(|E|)$, where $|E|$ denotes the number of edges in the task graph. Each instance of a 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 $O(|V|^3)$ [9], where $|V|$ denotes the number of tasks in the task graph. Therefore, the complexity of scheduling of one instance is $O(n|V|^3)$. When calculating the start time of an instance on each PE, $TDFS$ checks whether its predecessors need to be tested or not. The complexity of this check is $O(n|V|^2)$. Also, the computational complexity of scheduling one test is $O(|V|^2)$. Since $|E| < |V|^2$ and the number of task is $|V|$, the complexity of $TDFS$ is $O(|V|^4)$, given that n is fixed.

3.2.5 Illustrative Example

Figures 3 and 4 illustrate how Algorithm $TDFS$ works. In this example, we assume that the number of PEs, n, is four and that the task graph shown in Figure 3(a) is given. The set of tasks is partitioned into six task groups G_1, G_2, $\cdots$, G_6. Tasks in each task group are ordered according to their priorities as follows.

$$\begin{array}{llllll} G_1: & v_1 & G_2: & v_2 & G_3: & v_3, v_4 \\ G_4: & v_5, v_6 & G_5: & v_7, v_8 & G_6: & v_9 \end{array}$$

These task groups are ordered according to their heights. Then the Basic algorithm is applied to each task group in order. The task group whose height is the largest is selected first.

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, and its instance is tagged with "0". This is done by applying Procedure TDP to each PE. For example, an instance of v, which is indicated by v_4^0 in Figure 3, is scheduled as follows. As shown in Figure 3(b), an instance of v_3 (indicated by v_3^0) has already been assigned to p_0. It can be seen that the start times of v_4^0 on p_1 and on p_2 are 10 and 7, respectively. The start time of v_4^0

would be 6 on p_3 if no instances were duplicated, as shown in Figure 3(c). (Note that v_4 must receive necessary data from v_1.) In order to improve the start time of v_4^0, TDP applies *task duplication*. Figures 3(c) and (d) illustrate the concept of task duplication. In this case, TDP duplicates v_1 and schedules another instance to p_3 at time 0. (All instances generated in Step 1 are tagged with "0".) As a result of this duplication, v_4^0 can receive necessary data directly from v_1 without any communication delay, and the start time of v_4^0 on p_3 becomes 4. As a result, p_3 can start execution of v_4^0 earlier than p_1 and p_2. Therefore, v_4^0 is scheduled to p_3 as shown in Figure 3(d).

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. For example, since an instance of v_4 (v_4^0) is already scheduled to p_3 in Step 1, TDP is applied to p_1, p_2, and p_4. As a result, an instance tagged with "1" (v_4^1) is scheduled to p_4. (All instances generated in Step 2 are tagged with "1".) Similarly, each remaining task is scheduled so as to be executed on two different PEs as shown in Figure 3(e).

The Basic algorithm is applied to the remaining task groups G_4, G_5 and G_6. As a result, a schedule is obtained as shown in Figure 3(f).

When TDP calculates the start time of an instance on a PE, $TDFS$ also checks whether the predecessors of the instance need to be tested or not. For example, when an instance of v_7 with version number "0" (v_7^0) is scheduled to p_2, the algorithm decides to test four tasks; namely, v_1, v_2, v_3 and v_5, because v_7^0 receives data directly from v_5^1, and for v_1, v_2, v_3, which are all predecessors of v_7, their instances tagged with "1" are already assigned to p_2 (v_1^1, v_2^1, v_3^1).

Finally, TQ becomes $\{v_1, v_2, v_3, v_5, v_9\}$, and tests for these tasks are scheduled. In this example, since v_1, which is a common predecessor to the tasks in TQ, is assigned to all PEs, every test is duplicated and assigned to two distinct PEs. As a result, a 1-fault-secure schedule is obtained as shown in Figure 4(b).

4. Correctness Proof of Proposed Algorithms

In this section, we present a sketch of the correctness proof of Algorithm $TDFS$. For complete proofs of Algorithms STR and $TDFS$, readers are referred to [5].

In the following proofs, we let S denote a schedule generated by $TDFS$. As in [3, 4], we introduce an MVC_DAG $G' = (V', E')$ for S, where V' is the set of nodes and E' is the set of edges. G' represents the state of data exchanges between instances in S, i.e., G' is unique to S. Each node represents an instance of a task in S. If an instance s' receives necessary data from another instance s, then there is an edge from s to s'. If there exists a path from s' to s, we call s' an *ancestor* of s. Figure 4(c) shows an example of an

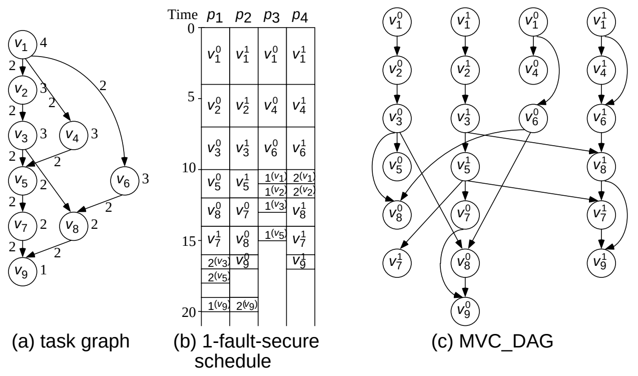


Figure 4. Example of MVC_DAG.

MVC_DAG for the schedule in Figure 4(b).

Lemma 1 For every fault pattern P' and for every interpretation I consistent with P', if a task v is tested in S, then the following conditions hold.
[Case 1:] If there exists only one instance t of the test in S, the following two conditions hold for every pair of instances, s and s', of v, where s and s' have different version numbers.

1. if $Label_I(s) \neq Label_I(s')$, then the test reliably reports "*not equal*", i.e., $Label_I(t) =$ "*n*".
2. if the outcome of the test is unreliable, then $Label_I(s) = Label_I(s') =$ "*c*".

[Case 2:] If there are two instances of the test in S, the following condition holds for every pair of instances, s and s', of v, where s and s' have different version numbers.

1. if $Label_I(s) \neq Label_I(s')$, then one of the two instances of the test, t_1 or t_2 reliably reports "*not equal*", i.e., $Label_I(t_1) =$ "*n*" or $Label_I(t_2) =$ "*n*".

Proof: If $Label_I(s) \neq Label_I(s')$, then there exists at least one ancestor of s or s', say x, in P'. In Case 1, the algorithm never assigns the test to PEs where s, s', or their ancestors have been scheduled. Therefore, $t \notin P'$, that is, the outcome of t is reliable. Consequently, t reliably reports "*not equal*". If t is not reliable, i.e., $t \in P'$, then for the same reason mentioned above, the labels of all the instances of v and its ancestors must be "*c*". That is, $Label_I(s) = Label_I(s') =$ "*c*".

In Case 2, there are two instances of the test in S. Each of them has been assigned to a different PE. Since only one PE is assumed to be faulty, it is obvious that either $t_1 \notin P'$ or $t_2 \notin P'$. Therefore, if $Label_I(s) \neq Label_I(s')$, then either t_1 or t_2 reliably reports "*not equal*". □

Lemma 2 Let s and s' be two instances of a task v, each with a different version number, and let $D(s) = (\alpha_1, \alpha_2, \cdots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \cdots, \alpha'_r)$. For every fault pattern

P' and for every interpretation I consistent with P', if $Label_I(s) = Label_I(s') \neq$ "c", then the following conditions hold.
[Case 1:] If $s \notin P'$ and $s' \notin P'$, then $Label_I(\alpha_q) = Label_I(\alpha'_q)$ for all q $(1 \leq q \leq r)$ and there exists at least one p $(1 \leq p \leq r)$ such that $Label_I(\alpha_p) = Label_I(\alpha'_p) \neq$ "c".
[Case 2:] If $s \in P'$ and $s' \notin P'$, then $Label_I(\alpha'_p) \neq$ "c" holds for some p $(1 \leq p \leq r)$.

Proof: In Case 1, from conditions (A), (D) and (E) in Definition 2, it is clear that $Label_I(s) = Label_I(s') \neq$ "c" if and only if the condition described above holds. In Case 2, it is clear from condition (A) in Definition 2 that if $Label_I(s') \neq$ "c", then $Label_I(\alpha'_p) \neq$ "c" for some p $(1 \leq p \leq r)$. (Note that $s \in P'$ and $s' \in P'$ never hold simultaneously.) □

Lemma 3 Let s and s' be two instances of a task v, each with a different version number. For every fault pattern P' and for every interpretation I consistent with P', if $Label_I(s) = Label_I(s') \neq$ "c", then there exists an instance t of some test in S such that $Label_I(t) =$ "n".

Proof: We prove this by induction.

[Base Step] From the definition of height, no tasks in the task group G_1 have immediate predecessors. Therefore, if $v \in G_1$, then $Label_I(s) = Label_I(s') =$ "c" or $Label_I(s) \neq Label_I(s')$ holds. Next, suppose $v \in G_2$ and $Label_I(s) = Label_I(s') \neq$ "c". When $s, s' \notin P'$, by Lemma 2, there is an instance α such that $Label(\alpha) \neq c$ and α is both in $D(s)$ and in $D(s')$. In this case, whichever version number α is tagged with, $TDFS$ guarantees that the task corresponding to α is tested. Lemma 1 also ensures that the test for α reliably reports "*not equal*". When $s \in P'$ and $s' \notin P'$, s' receives data from an instance assigned to the same PE as s. Due to the rule of assigning tests, a test is assigned for this instance, and by Lemma 1 it reliably reports "*not equal*".

[Induction Step] Assume that the lemma holds if $v \in G_1 \cup G_2 \cup \cdots \cup G_k (k \geq 2)$. Now suppose $v \in G_{k+1}$ and $Label_I(s) = Label_I(s') \neq$ "c". Then two cases must be considered, namely, [Case 1:] $s \notin P'$ and $s' \notin P'$, and [Case 2:] $s \in P'$ and $s' \notin P'$. For Case 1, we can prove that there is a test that reports "not equal" by Lemma 1 and Lemma 2. For Case 2, we can prove this by Lemma 1. Thus the lemma follows. (For a complete proof, see [5].) □

Theorem 1 $TDFS$ generates 1-fault-secure schedules.

Proof: Suppose that an instance of an output task v is labelled with an erroneous value. Then, by Lemma 3, there is a test that reports "not equal", or there is another instance of v that is labelled with "c". Also in the latter case, there is a test that reliably reports "*not equal*" by Lemma 1, since all output tasks are tested. Thus the theorem follows from the definition of a 1-fault-secure schedule. □

5. Experimental Evaluation

5.1. Simulation Environment

Using a large number of task graphs as a workload, we performed simulations for comparison studies of Algorithms STR and $TDFS$. In the simulations, we used task graphs for two practical parallel computations: Gaussian elimination [10] and LU-decomposition [11]. These task graphs can be characterized by the size of the input matrix because the numbers of tasks and edges in the task graph depends on the 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 sizes 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 [10].

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

In the simulation, for each task graph, the execution time of a test is set to the smallest execution time of tasks, and the communication cost between a test and tasks is set to the average communication delay between tasks.

As a baseline, we used the finish time of a (non-fault-secure) schedule generated by DSH. All results presented in this section are normalized to this length. In the studies, we considered two cases: the number of PEs $n = 8$, and $n = 16$.

5.2. Evaluation Results

Figures 5 and 6 show the simulation results for Gaussian elimination and LU-decomposition task graphs, respectively. The value of *ccr* is fixed to 5, and the matrix size is varied so that the number of tasks in the corresponding task graph ranges from 100 to 400. The results show that $TDFS$ outperforms STR. As the matrix size increases, the difference between the performance of $TDFS$ and that of STR increases. The following reason is conjectured. In general, as the size of the task graph increases, its parallelism also increases (here, parallelism means the maximum number of tasks that can be executed in parallel at a time). We can take advantage of the 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, $TDFS$ can extract more parallelism than STR because STR can essentially use only $n/2$ PEs.

Figures 7 and 8 show the simulation results when the matrix size is 24. In this simulation, we varied the value of ccr from 0.1 to 10.0. In both kinds of task graphs, when the value of ccr is small, e.g., $ccr < 1.0$, STR shows better performance than $TDFS$. In most cases, the number of tests in a schedule obtained by STR is smaller than $TDFS$ because STR requires tests for the output tasks only. Note that as communication delays decrease, the amount of idle time between tasks, which is available for scheduling tests by $TDFS$, decreases. As a result, $TDFS$ has worse performance than STR. On the other hand, when $ccr \geq 1.0$, $TDFS$ shows better performance than STR.

Compared with the non-fault-secure scheduling algorithm DSH, $TDFS$ achieved 1-fault security at the cost of a small increase in schedule length. For example, in the case where $n = 16$ and $ccr \geq 1$, $TDFS$ achieved 1-fault security with less than 20% overhead. (Note that each result is normalized to the schedule length of DSH.)

6. Conclusions

In this paper, we proposed two multiprocessor scheduling algorithms, STR and $TDFS$, to achieve 1-fault security. We showed that the time complexity of these algorithms is $O(|V|^4)$, where $|V|$ is the number of tasks in the given task graph.

We performed simulation studies using two kinds of task graphs for practical parallel computation; namely, Gaussian elimination and LU-decomposition. As a result, it was found that $TDFS$ outperforms STR especially when the value of ccr is large ($ccr \geq 1.0$).

A drawback of $TDFS$ is its running time. For example, for a Gaussian elimination task graph with matrix size = 24 (the number of tasks is 297), it took 8391 seconds to produce a 1-fault-secure schedule. (We conducted the simulations on a COMPAQ XP1000 workstation.) We consider improving the running time as future work.

References

[1] P. Banerjee and J. A. Abraham, "Fault-secure algorithms for multiple processor systems," Proc. of *11th Int'l Symp. on Computer Architecture*, pp. 270-287, 1984.

[2] H. El-Rewini, H. H. Ali, and T. Lewis, "Task scheduling in multiprocessing systems," *IEEE Computer*, vol. 28, no. 12, pp. 27-37, 1995.

[3] D. Gu, D. J. Rosenkrantz, and S. S. Ravi, "Construction and analysis of fault-secure multiprocessor schedules," Proc. of *21th IEEE Int'l Symp. on Fault-Tolerant Computing*, pp. 120-127, 1991.

[4] D. Gu, D. J. Rosenkrantz, and S. S. Ravi, "Fault/error models and their impact on reliable multiprocessor schedules," Proc. of *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 176-184, 1992.

[5] K. Hashimoto, "Multiprocessor scheduling algorithms for high reliability," PhD dissertation, Dept. of Informatics and Mathematical Science, Osaka University, 2000.

[6] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "A multiprocessor scheduling algorithm for low overhead fault-tolerance," Proc. of *17th IEEE Int'l Symp. on Reliable Distributed Systems*, pp. 186-194, 1998.

[7] B. W. Johnson, "Design and Analysis of Fault-Tolerant Digital Systems," Addison-Wesley, 1989.

[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, 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.

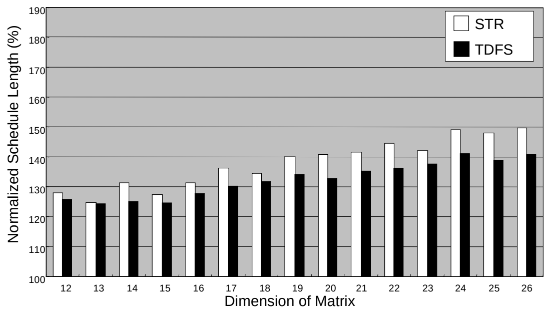
[10] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, 1996.

[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, 1983.

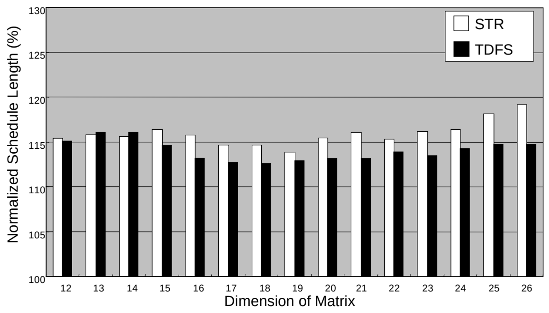
[12] K. R. Pattipati, T. Kurien, R. -T. Lee, and P. B. Luh, "On mapping a tracking algorithm onto parallel processors," *IEEE Trans. Aerospace and Electronic Systems*, vol. 26, no. 5, pp. 774-791, 1990.

[13] 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, 1995.

[14] J. Wu, E. B. Fernandez, and D. Dai, "Optimal fault-secure scheduling," *Computer Journal*, vol. 41, no. 4, pp. 208-222, 1998.

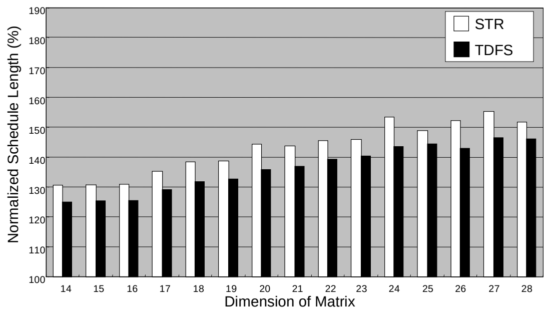


(a) # of PEs = 8

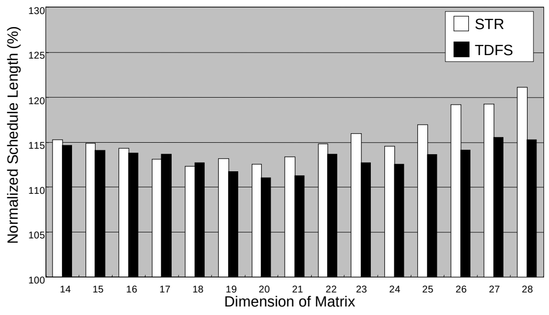


(b) # of PEs = 16

Figure 5. Results for Gaussian elimination task graphs.

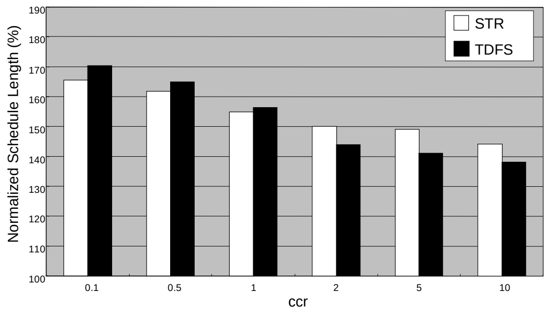


(a) # of PEs = 8

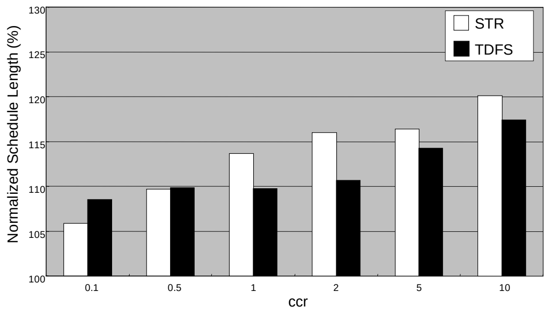


(b) # of PEs = 16

Figure 6. Results for LU-decomposition task graphs.

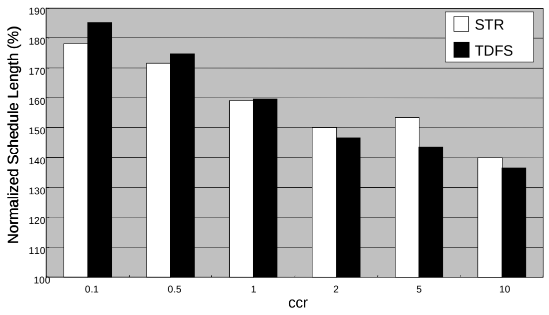


(a) # of PEs = 8

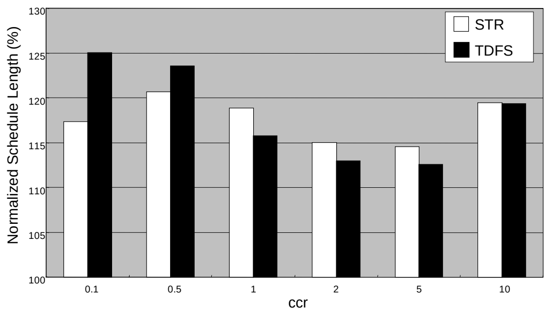


(b) # of PEs = 16

Figure 7. Results for Gaussian elimination task graphs.



(a) # of PEs = 8



(b) # of PEs = 16

Figure 8. Results for LU-decomposition task graphs.