J Grid Computing (2015) 13:507–525
DOI 10.1007/s10723-015-9331-1

Fault-Tolerant Dynamic Rescheduling for Heterogeneous Computing Systems

Jing Mei · Kenli Li · Xu Zhou · Keqin Li

Received: 3 September 2014 / Accepted: 25 March 2015 / Published online: 14 April 2015

Abstract As the scale and complexity of heterogeneous computing systems grow, failures occur frequently and have an adverse effect on solving large-scale applications. Hence, fault-tolerant scheduling is an imperative step for large-scale computing systems. The existing fault-tolerant scheduling algorithms belong to static scheduling, and they allocate multiple copies of each task to several processors no matter whether processor failures affect the execution of tasks. Such active replication strategies not only waste resource but also sacrifice the makespan. What is more, they cannot guarantee the successful execution of applications. In this paper, we propose a fault-tolerant dynamic rescheduling algorithm named FTDR, which can overcome above drawbacks. FTDR keeps listening to the processor failure, and reschedules the suspended tasks once failures occur. Because FTDR reschedules the tasks that are suspended because of failures, it can tolerate an arbitrary number of failures. Randomly generated DAGs are tested in our experiments. Experimental results show that the proposed algorithm achieves good performance in terms of makespan and resource consumption compared with its direct competitors.

Keywords Directed acyclic graph · Fault tolerance · Heterogeneous computing system · Task reschedule

J. Mei · K. Li (✉) · X. Zhou · K. Li
College of Information Science and Engineering, Hunan University, and National Supercomputing Center in Changsha, Hunan, 410082, China
e-mail: lkl@hnu.edu.cn

J. Mei
e-mail: jingmei1988@163.com

X. Zhou
e-mail: happypanda2006@126.com

K. Li
e-mail: lik@newpaltz.edu

K. Li
Department of Computer Science, State University of New York, New Paltz, New York 12561, USA

1 Introduction

A heterogenous computing (HC) system consists of diverse sets of resources interconnected with a high-speed network, which can support executing computationally intensive parallel applications with diverse computing needs. To achieve the efficient execution of applications, many scheduling strategies are proposed, for example, list scheduling [1–5], duplication [6–12], clustering [13–15] and so on. The goal of those algorithms is to achieve a good mapping of tasks to processors, minimizing the schedule length (makespan). Makespan is defined as the time difference between the start and finish of a sequence of jobs or tasks. However, along with the increasing number of processors in systems, the reliability is declined. The reliability of a system is referred to as the probability that it has no failure under stated conditions for a specified amount of time [16]. Nowadays, a heterogenous system such as the supercomputers has more faults than before, so fault tolerance has become an important role in improving the performance of the systems.

There are two major failure types, transient and permanent, and they are assumed to be independent. In a nutshell, transient failures invalidate only the execution of the current task. The processors subject to that failures will be able to recovery after a period of time and execute the subsequent tasks assigned to it. On the contrary, permanent failures are unrecoverable. Once the failure occurs, the corresponding processor is down until the end of the whole execution. If the algorithms are not fault-tolerant, the applications may be completed unsuccessfully on the scheduling of them. In the past decade, there are an increasing literatures that focus on developing techniques to achieve fault tolerance. One of the fault-tolerant techniques is replication strategy. Replication strategy is widely used in many systems such as the distributed storage [17] and Hadoop [18]. In such systems, replication strategy can efficiently improve the security of data and shorten the response time of application execution by sacrificing a small amount of resources. However, when replication strategy is adopted in fault tolerance, it has several drawbacks. In existing works, each task is executed with multiple replicas to guarantee the reliability of application execution, and the number of replicas of each task is not small, which leads to a great amount of extra resource overhead and longer response time. Whereas, the great resource overhead and the sacrifice of performance cannot absolutely guarantee the successful execution of applications. Hence, adopting replication strategy to improve reliability and tolerate fault is not recommended.

Another kind of fault-tolerant mechanism is checkpointing [19, 20]. It achieves fault tolerance by periodically saving the state of a process during the failure-free execution and restoring the system back to a consistent state after a failure. This mechanism can improve system reliability but incurs much additional time on saving checkpoints and rollback-recovery [21]. The additional time may reach or even exceed the MTTF (mean time to failure) when the system performance sits between the petascale and exascale levels. Moreover, the rollback recovery is complicated because message induces inter-process dependencies. Once a processor fails, all the processors should recover to a consistent global checkpoint. Hence, checkpointing is not a good mechanism for peta/exascale supercomputers and distributed systems.

In this paper, we introduce a fault-tolerant scheduling algorithm which is based on dynamic reschedule strategy, called Fault-Tolerant Dynamic Rescheduling (FTDR) algorithm. FTDR is a dynamic event-driven scheduling algorithm. It keeps listening and responses to five kinds of events, including application submission event, task start event, task completion event, processor failure event, and processor recovery event. Once a processor failure is detected, the scheduler reassigns new processors for the tasks which are located on the failure processor. Because FTDR reschedules tasks interrupted by failures, it can tolerate an arbitrary number of failures. Experimental results demonstrate that the proposed algorithm FTDR does neither sacrifice makespan nor waste resource compared with its direct competitors, FTSA [22] and MaxRe [23].

The paper is organized as follows. Section 2 presents a brief review of the literatures focusing on fault-tolerant scheduling. Section 4 gives an example to illustrate the motivation of the paper. In Section 3, the models used are presented. And then the proposed algorithm FTDR (Fault-Tolerant Dynamic Rescheduling algorithm) is described in detail in Section 5. Section 6 presents the experimental results of FTDR compared with the other three algorithms and the detailed analysis is given. Finally, Section 7 concludes the work.

2 Related Work

In past decade, there is a lot of research focusing on fault-tolerant scheduling problem, only with different emphasis. In the earlier studies, the concept of system reliability was proposed. System reliability is defined as the probability that the system can run an entire task successfully. Many scheduling algorithms are addressed based on reliability analysis [24–32]. Shatz et al. [24] designed an algorithm that maximizes the system reliability. In its model, failures from processors and communication links are considered time-dependent and treated evenly, and an explicit cost function is provided to measure system reliability. Qin and Jiang [25] designed a reliability-driven scheduling algorithm for parallel real-time tasks which aims at meeting the respective deadlines of all the subtasks while maximizing reliability. Dongarra et al. [26] presented two algorithms that optimize both makespan and reliability. The first algorithm in [26] is to maximize the reliability subject to makespan minimization. And the second one is to trade off between reliability maximization and makespan minimization based on the product *failure rate* × *unitary instruction execution time*. In [33], a weighted bi-objective list scheduling algorithm (BSA) is proposed. BSA firstly sorts the tasks in non-increasing order of their priority, and then chooses the processor that minimizes a weighted integrated cost function. In the objective function, both of makespan and reliability are normalized by their maximum values. Girault et al. [28] proposed an algorithm which also takes into account both of makespan and reliability. The algorithm is allowed to produce several trade-off solutions, among which the user can chose the solution that best fits the requirements. The reliability-driven scheduling algorithm proposed by Tang et al. in [29] introduced reliability priority rank to estimate the task's priority by considering reliability overheads. Jeannot et al. [31] first showed that the two objectives, minimizing makespan and maximizing reliability, are contradictory, and then designed a $(1+\epsilon,1)$-approximation algorithm of the Pareto-front. Obviously, the reliability achieved by these algorithms is limited. And to achieve higher reliability, special schemes such as active replication are necessary.

The primary and backup scheduling algorithm can tolerate one failure in the systems. It is first proposed to solve the fault-tolerance scheduling problem of independent tasks in real-time systems, but then many literatures apply the strategy to solve the fault-tolerance scheduling problem of tasks with precedence constraints [34–36]. Qin and Jiang [34] proposed the eFRD algorithm. In [34], the emphasis is to analyze the necessary conditions for tasks' backup copies to safely overlap in time with each other. Zheng and Veeravalli proposed three primary and backup scheduling algorithms in [35]. First two algorithms are designed to minimize response time and replication cost respectively. The third one is to minimize replication cost while not affecting response time. In [36], Zheng et al. performed the further research on primary-backup approach. It is concluded that two important constraints must be satisfied when scheduling the backup copies. Based on the conclusion, two algorithms, MRC-ECT and MRC-LRC, are proposed to schedule backups of independent tasks and dependent jobs, respectively. These works introduced above only can tolerate one failure, which is far from enough for scheduling problem.

Considering crash failures, the active replication scheme is incorporated into the scheduling algorithms [22, 23, 37]. The FTSA algorithm proposed in [22] is an extended version of the classic HEFT algorithm [2]. It allocates $\varepsilon + 1$ copies of each task to different processors to tolerate an arbitrary number ε of fail-silent processor failures. Zhao et al. pointed out in [23] that FTSA leads to significant resource consumption, hence they proposed the MaxRe algorithm in which the reliability analysis is incorporated into the active replication scheme to exploit a dynamic number of replicas for different tasks. In [37], Benoit et al. took the contention into account and put more emphasis on the practical one-port communication model. However, these algorithms waste a great of resource and sacrifice the makespan. What's more, they cannot guarantee the execution success of applications.

3 Models

The scheduling system model discussed in this paper consists of a target computing platform and an application. In this section, we first introduce the computing system model in detail. And then, we describe the model of parallel application with precedence constraints. Lastly, we outline the architecture of the fault-driven scheduling.

3.1 Computing System Model

This paper researches the scheduling problem of applications on heterogeneous computing systems. Homogenous computing system is a special situation of heterogeneous ones. Let $P = \{p_i | 0 \le i \le m-1\}$ be a set of m processors with different capacities. The capacity of a processor processing tasks depends on how well the processor architecture matches tasks' processing requirements. A task scheduled on its best-suited processor will spend shorter execution time than on a less-suited processor. The best processor for one task may be the worst one for another task. This type of model is described by [38] and used in [2, 6, 12, 39]. The computers in the system are connected by a shared bus with unlimited channels, hence, the communication contention and the end-point contention are not taken into account here. Every channel has independent buffer to store the data and initiates the communication from that buffer when it finishes the ongoing communication [40]. The communication model adopted in this paper is broadcast communication model. In this model, a channel carrying a communication delivers the data to all the processors [40].

Processors are subject to failures during the execution of the tasks that are assigned to them. There are two main categories of failures which may occur during the execution of a task on a processor: transient failure and permanent failure, which are introduced in [22]. In a nutshell, transient failures invalidate only the execution of the current task. The processors subject to that failures will be able to recovery after a period of time and execute the subsequent tasks assigned to it. On the contrary, permanent failures are unrecoverable. Once the failure occurs, the corresponding processors is down until the end of the whole execution. Based on the common exponential distribution assumption in the reliability research [41, 42], for each processor $p_i \in P$, the arrival of failures following a Poisson distribution with λ_i. λ_i is a positive real number, and equals to the expected number of occurrence of failures in unit time t. The processors failure distribution in unit time t can be represented as:

$$f(k, \lambda_i) = \frac{\lambda_i e^{-\lambda_i t}}{k!}, \tag{1}$$

where k is the number of occurrences of failures in unit of time. The reliability of processors is defined as the probability that no failure occurs in unit of time t which is calculated as

$$f(k=0, \lambda_i) = e^{-\lambda_i t}. \tag{2}$$

Suppose the expected numbers of failures in unit time for all processors are $\Lambda = \{\lambda_i | 0 \leq i \leq m-1\}$, and different processor failures are always supposed to be independent.

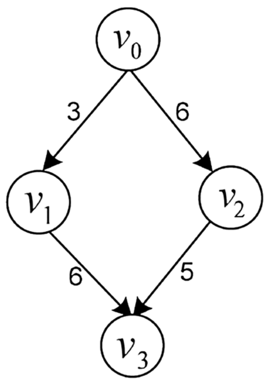


Fig. 1 A simple DAG representing precedence-constraint application graph

3.2 Application Model

A parallel application with precedence constraints can be represented by a *Directed Acyclic Graph* (DAG) $G(V, E, W, C)$, which consists of a set of nodes $V = \{v_i | 0 \leq i \leq n-1\}$ representing the tasks of the application and a set of directed edges E representing dependencies between tasks. An edge $e_{ij} \in E$ from node v_i to v_j ($v_i, v_j \in V$) represents v_j receiving data from v_i, where v_i is called a parent of v_j and v_j is a child of v_i. The positive weight $w(v_i, p_k) \in W$ represents the computation time of task v_i on processor p_k for $0 \leq i \leq n-1$ and $0 \leq k \leq m-1$. The nonnegative weight $data(e_{ij})$ associated with edge $e_{ij} \in E$ represents the volume of data transferred from v_i to v_j. Let μ be the data transfer rate between processors and L_m be the communication startup time. Thus, the communication time from task v_i to task v_j is calculated as follows:

$$c(e_{ij}) = L_m + \frac{data(e_{ij})}{\mu} \tag{3}$$

In our paper, we set $\mu = 1$ and $L_m = 0$ for convenience, so $c(e_{ij}) = data(e_{ij})$ and the output data of tasks can be transferred to the cache in the channels without delay.

Table 1 Computation cost matrix W

Task node	p_0	p_1	p_2	$\overline{w_i}$
v_0	6	5	7	6
v_1	11	9	8	9.33
v_2	8	12	8	9.33
v_3	3	4	3	3.33

We show a simple DAG in Fig. 1 which consists of seven nodes, and Table 1 lists the computation cost matrix W of three processors. The average computation cost of task v_i is defined as $\overline{w_i} = \frac{\sum_{k=0}^{m-1} w(v_i, p_k)}{m}$. The cost model has been adopted in many works [1–3, 6, 7]. The computation and communication costs are the priori knowledge required to generate a proper schedule. Commonly, these costs are measured in time and they are obtained by instruction analysis or estimation. In our paper, we assume that these costs can be measured accurately. When considering the inaccurate priori knowledge, the fault-tolerant scheduling problem is much more complicated, hence, we do not consider this situation.

For task v_i, all its direct parent nodes are denoted as $Pare(v_i)$ and all its direct child nodes are denoted as $Child(v_i)$. A task $v_i \in V$ having no parents, $Pare(v_i) = \phi$, is called *entry task*, such as task v_0 in Fig. 1. A task having no children, $Child(v_i) = \phi$, is called an *exit task*, such as v_3. A DAG may have multiple entry tasks and multiple exit tasks.

3.3 Fault-driven Scheduling Architecture

Figure 2 depicts the fault-driven scheduling architecture in a heterogeneous distributed environment. In the architecture, the global scheduler works with the processor supervisor cooperatively.

In the scheduler, all subtasks of an application and the corresponding information are submitted to the master processor by a special user who is of the authority. The global scheduler in the master schedules the subtasks. A *priority queue* (PQ) for the arriving application is determined by the scheduler according to the submitted information. For each task in the queue, the global scheduler determines a proper assignment scheme, including which processor it is assigned to and when the execution starts and so on. The schedule information is stored in the schedule queue (SQ). The status of each processor is supervised by the processor supervisor. In this architecture, the heartbeat mechanism can be adopted to detect the processor failures. Each processor sends "heartbeat" periodically to the processor supervisor in the master node. If the supervisor doesn't receive the "heartbeat" signal from a processor for three periods, the processor is considered to be failed. When a failure is detected, the processor supervisor notices the scheduler to respond for it.

4 A Motivational Example

To illustrate the motivation of this research, we first present an example in this section. Figure 3 gives two schedules of the application shown in Fig. 1, which are generated by the HEFT and MaxRe algorithms, respectively.

Figure 3a is the schedule generated by the HEFT algorithm. HEFT is a classical static list scheduling algorithm, which is famous for its low complexity. According to the generated schedule, tasks v_0 and v_1 are assigned to processor p_1, and tasks v_2 and v_3 to processor p_0. Both of v_1 and v_2 receive data from v_0

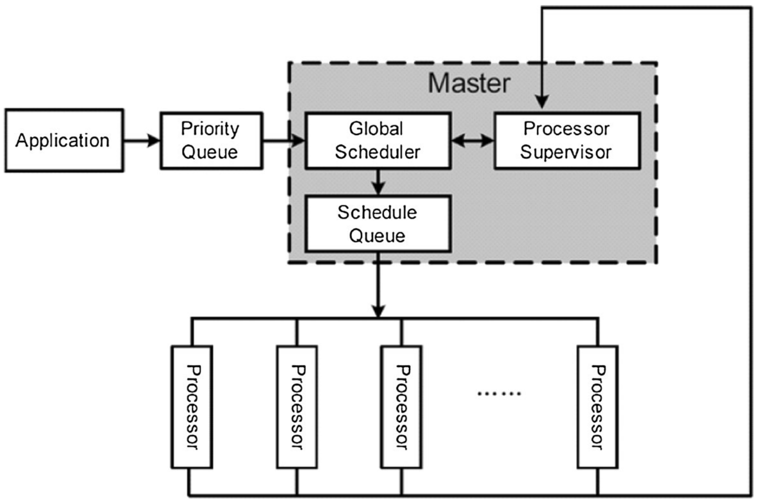


Fig. 2 The fault-driven scheduler model for dynamic scheduling of parallel application in heterogeneous environment

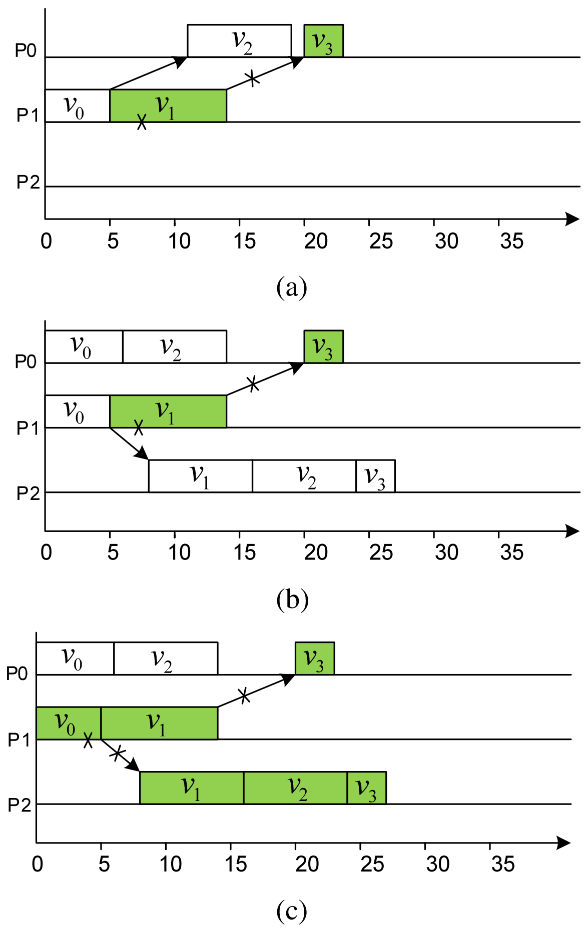


Fig. 3 A motivational example: **a** HEFT with a failure; **b** Application is executed successfully using MaxRe when a failure occurs; **c** Application fails using MaxRe when a failure occurs. The cross represents a failure occurs at the instant and the shaded tasks represent that the tasks fail

and v_3 receives data from v_1 and v_2. The makespan of HEFT is 23.

However, as a static scheduling algorithm, HEFT cannot tolerate fault. A failure occurring during the execution of a task will lead to the failure of the whole application. Figure 3a shows a scenario that a failure occurs on processor p_1 at time seven. The task v_1 fails due to the failure. Consequently, its child v_3 cannot be executed. Therefore, the schedule generated by HEFT is unavailable under one failure.

Figure 3b and c give the schedules generated by the MaxRe algorithm. MaxRe is a fault-tolerant scheduling algorithm based on the reliability analysis. It schedules multiple replicas for each task to improve the system reliability. From Fig. 3b, we can see that each task has two replicas in the schedule and the application can be executed successfully using MaxRe when a failure occurs at the same time as Fig. 3a.

However, the schedule generated by MaxRe cannot completely guarantee the success execution of applications. Figure 3c is a scenario that a failure leads to the application's failure. When a failure occurs at instance four on processor p_1, the replica of v_0 on p_1 fails. According to the MaxRe algorithm, two replicas of v_1 on p_1 and p_2 should receive data from the paused replica of v_0, so they cannot be executed due to the failure. Consequently, the whole application fails.

From above example, we know that, static scheduling algorithms have poor adaptivity for fault-tolerance. Even though the replication strategy can improve the reliability greatly, it still cannot completely guarantee the successful execution of application. Moreover, it degrades the performance in terms of makespan and leads to a great amount of resource waste at the same time. To overcome the drawbacks of static scheduling algorithm, we propose a fault-tolerant dynamic rescheduling algorithm in the study.

5 The Proposed Algorithm

In this section, the fault-tolerant dynamic rescheduling algorithms, **FTDR** for short, are presented for heterogeneous computing systems. Before describing the FTDR algorithm in detail, we first introduce some basic notations.

5.1 Some Basic Notations

A schedule of an application associates a processor and a starting time with each task within the application DAG. In a generated schedule, if processor p_k is assigned to execute task v_i, we say that p_k is the *expected assigned processor* of v_i, denote by $ap(v_i) = p_k$. The *expected start time* of v_i is denoted by $st(v_i)$. The scheduler dispatches tasks to the corresponding processors according to the generated schedule. For a static scheduling algorithm, the expected assigned processor of task v_i must be its executed processor. However, considering the processor failures, the executed processor of v_i is not always the expected assigned processor. We define the executed processor of v_i as its *actual executed processor*, denoted as $aep(v_i)$.

In order to determine the expected assigned processor and the expected starting time for each task of a given application, it is necessary to define est and eft firstly. $est(v_i, p_k)$ and $eft(v_i, p_k)$ denote the earliest execution start time and the earliest execution finish time of task v_i on processor p_k, respectively. For the entry task v_{entry},

$$est(v_{entry}, p_k) = 0. \tag{4}$$

For the other tasks, the est values are calculated recursively. In order to calculate the est of a task v_i, its all immediate parent tasks must have been scheduled.

Definition 5.1 The **data arrive time (dat)** of task v_i on processor p_k, denoted by $dat(v_i, p_k)$, is the instant that v_i has received data from all of its parent tasks when it is assigned to p_k.

Let $Child(v_i)$ be the child task set of v_i. For each $v_j \in Child(v_i)$, let $aft(v_j)$ be the *actual finish time* of v_j. Then, the data arrive time of v_i is calculated by

$$dat(v_i, p_k) = \max_{v_j \in Pare(v_i)} \{aft(v_j) + c(e_{ji})[aep(v_j) \neq p_k]\}, \tag{5}$$

where $[aep(v_j) \neq p_k]$ is equal to 1 if $aep(v_j) \neq p_k$, otherwise 0. Apart from the dat, the est of v_i also depends on the status of p_k. If there are tasks assigned to p_k, v_i must wait until p_k becomes idle. Hence, the est of task v_i on p_k is obtained by

$$est(v_i, p_k) = \max\{dat(v_i, p_k), avail(p_k)\}, \tag{6}$$

where $avail(p_k)$ is the instant that processor p_k becomes idle.

Hence, the eft of task v_i on p_k is obtained by

$$eft(v_i, p_k) = est(v_i, p_k) + w(v_i, p_k). \tag{7}$$

When determining the assignment for each task, different scheduling algorithms have their different methods to select the assigned processor. In this paper, task is assigned to the processor which leads to a minimal eft, which means,

$$ap(v_i) = p_k, where\, eft(v_i, p_k) = \min_{p_j \in P}\{eft(v_i, p_j)\}. \tag{8}$$

The expected start time of task v_i on $ap(v_i)$ is obtained by

$$st(v_i) = est(v_i, ap(v_i)). \tag{9}$$

The expected finish time $ft(v_i)$ of task v_i on $ap(v_i)$ is calculated by

$$ft(v_i) = eft(v_i, ap(v_i)). \tag{10}$$

Further, the schedule length, also the makespan of the application is obtained by

$$makespan = \max_{v_i \in V}\{ft(v_i, aep(v_i))\}. \tag{11}$$

The notations used in this paper are summarized in Table 2.

5.2 Task Priority

To ensure that the generated schedule satisfies the precedence constraints between tasks, tasks are attached with scheduling priorities which are based on upward ranking. The *upward rank* of task v_i is recursively calculated as follow:

$$rank_u(v_i) = \overline{w_i} + \max_{v_j \in Child(v_i)}\{c(e_{ij}) + rank_u(v_j)\}. \tag{12}$$

where $Child(v_i)$ is the set of immediate child tasks of v_i, $c(e_{ij})$ is the communication cost of edge e_{ij}, and $\overline{w_i}$ is the computation cost of task v_i. The upward rank value of the exit tasks v_{exit} is equal to

$$rank_u(v_{exit}) = \overline{w_{exit}}. \tag{13}$$

Table 3 gives the upward rank of each task of DAG in Fig. 1. The task priorities mentioned above are only related to their schedule order rather than their actual execution order. These two kinds of order are different. The task which is scheduled later may be executed earlier by making use of idle period. In our algorithm, a schedule queue SQ is constructed in which all ready tasks are listed in the nonincreasing order of priorities. The schedule of each task in the ready queue is determined by the scheduler. At the right instance, the scheduler starts the communications and dispatches the tasks to the allocated processors.

Definition 5.2 A ready task is defined as the task whose parent tasks have completed their executions.

At the beginning, the schedule queue SQ consists of the entry tasks because they have no parent tasks. They are executed according to the generated schedule. When a task v_i is completed, SQ is updated by removing v_i from SQ and inserting new ready tasks

Table 2 Definitions of the notations

Notation	Definition
V	A set of n weighted tasks in the application
v_i	$v_i \in V$ representing the ith task in the application
E	A set of directed edges representing the constraint among tasks in V
e_{ij}	The constraint between tasks v_i to v_j
$c(e_{ij})$	The communication cost between tasks v_i and v_j
P	A set of m heterogenous processors
p_k	The kth processor
PQ	The priority queue consisting of all tasks in increasing order of $rank_u$.
SQ	The schedule queue, which records the schedule of the ready tasks.
$Pare(v_i)$	The set of immediate parents of task v_i
$Child(v_i)$	The set of immediate children of task v_i
$ap(v_i)$	The expected processor which v_i is allocated
$st(v_i)$	The expected start time of task v_i on processor $ap(v_i)$
$dat(v_i, p_k)$	The time that the data required by task v_i arrives at processor p_k.
$ft(v_i)$	The expected finish time of task v_i on processor $ap(v_i)$
$est(v_i, p_k)$	The earliest start time of task v_i on processor p_k
$eft(v_i, p_k)$	The earliest finish time of task v_i on processor p_k
$rest(v_i, p_k)$	The earliest start time of task v_i on processor p_k when v_i is rescheduled to p_k
$reft(v_i, p_k)$	The earliest finish time of task v_i on processor p_k when v_i is rescheduled to p_k

to SQ. The children's allocated processors and start times are calculated in the order of their priorities.

5.3 Global Scheduler Algorithm

A global scheduler is adopted in this scheduling algorithm. Since the algorithm is to deal with unexpected processor failures when tasks are executing, the global scheduler must make dynamic schedule when failures occur. The global scheduler is notified by some special events, and its functionality can be modeled by the following events.

1) Application submission event. When an application is submitted to the system, an application submission event is generated. At this event, the global scheduler initiates a priority queue and schedule queue. The ready tasks are scheduled and all the schedule information is stored in the schedule queue.
2) Task start event. According to the generated schedule, task start event will be generated at the start time of each task. At this event, the global scheduler dispatches the task to its allocated processor.
3) Task completion event. When a processor completes execution of the assigned tasks on it, it generates a task completion event. At this event, the global scheduler starts the outgoing communication of the finished task. The output data is transferred to the destination processors where the child tasks are assigned. Besides, the data is transferred to the master and stored as a backup.
4) Processor failure event. A processor failure event is generated when a failure occurs. At this event, the information of available processors is updated. The global scheduler reschedules the tasks interrupted by the failure, and the tasks as well as the backup data are reassigned to new processors.
5) Processor recovery event. A processor recovery event is generated when a failure processor is

Table 3 The upward rank of tasks

Task node	v_1	v_2	v_3	v_4
$rank_u$	29.333	18.666	17.666	3.333

recovery. At this event, the information of available processors is updated.

The global scheduler keeps listening to the five events during the period of applications execution. At the beginning, when an application is submitted to the system, the global scheduler initiates the priority queue PQ and generates an assignment for each ready task. The generated assignments are stored in a schedule queue SQ. According to the schedule, responding to each task start event, the scheduler dispatches the tasks to their allocated processors and the tasks start execution. When a task completes its execution, the scheduler schedules the required communications to appropriate processors. Meanwhile, the schedule queue SQ is updated by inserting new ready tasks and the scheduler generates schedule for each ready task in SQ. When a processor failure occurs during the task execution, the scheduler reallocates new processor for the paused task. During the following period, the failure processor cannot be allocated to execute any tasks until it is recovery. When a failure processor is recovery, the scheduler will consider the processor when scheduling the following tasks.

In the process, the scheduler starts the communication once a task is completed. The output data is transferred to all processors where its child tasks are assigned and the master as a backup. Once the child tasks have received the required data from all parents and the assigned processors are ready, the child tasks can start executing. Whereas, if a processor fails, the current task certainly cannot be completed on schedule. In order to ensure the execution success of applications, it is necessary to allocate new processor to the paused task. The event handling details are described in Algorithm 1.

Algorithm 1 Global scheduler for broadcast communication model

```
1: Initialize a priority queue $PQ$ and insert all tasks into
   $PQ$ in the nonincreasing order of $rank_u$
2: Initialize a schedule queue $SQ$ as empty
3: Initialize an available processor set $P_a$, $P_a = P$
4:
5: **Event-**Application submitted and task completed
6: **for** all ready tasks in $PQ$ **do**
7:     $v_i \leftarrow$the first ready task in $PQ$
8:     **for** each processor $p_j$ in $P_a$ **do**
9:         Calculate $est(v_i, p_j)$ and $eft(v_i, p_j)$
10:    **end for**
11:    $ap(v_i) \leftarrow$the processor $p_k$ with minimal $eft$ of $v_i$
12:    $st(v_i) \leftarrow est(v_i, ap(v_i))$, $ft(v_i) \leftarrow eft(v_i, ap(v_i))$
13:    Insert the schedule $\langle v_i, ap(v_i), st(v_i), ft(v_i)\rangle$ to $SQ$
14:    Remove $v_i$ from $PQ$
15: **end for**
16: **End Event**
17:
18: **Event-**Task start event {The start time of task $v_i$ arrives}
19: $v_i$ is dispatched to its allocated processor $ap(v_i)$ and
    starts executing
20: Remove the schedule of $v_i$ from $SQ$
21: **End Event**
22:
23: **Event-**Processor failure event {A processor $p_k$ fails and
    the paused task is $v_i$}
24: **for** each available processor $p_j$ in $P_a$ **do**
25:     Calculate $rest(v_i, p_j)$ and $reft(v_i, p_j)$
26: **end for**
27: $ap(v_i) \leftarrow$the processor $p_j$ that minimizes $reft$ of $v_i$
28: $st(v_i) \leftarrow rest(v_i, ap(v_i))$, $ft(v_i) \leftarrow reft(v_i, ap(v_i))$
29: Update the schedule information of $v_i$ in $SQ$
30: **End Event**
31:
32: **Event-**Processor recovery event {Processor $p_k$ is recovered}
33: Update the available processor set $P_a$
34: **End Event**
```

When the application is submitted to the system, the global scheduler generates the initial assignments of all entry tasks (5-16). The generated assignments are recorded in the schedule queue SQ, which consists of the allocated processor, the start time and the finish time of each ready task. The assignments of tasks are inserted into the schedule queue S (line 13).

When the start time of a task arrives, the global scheduler is triggered by a task start event, and dispatches the task to its allocated processor. When a task starts executing, its schedule information is removed from SQ (lines 18-21). When a task, see v_i, is completed, the new ready tasks are scheduled and the schedule information is recorded in SQ, too (lines 5-16). Meanwhile, once the task is completed, its output data is transferred to the destination processors and the master. When a processor failure occurs, the global scheduler makes a dynamic rescheduling strategy to deal with the failure (lines 23-30). First, the scheduler records the paused task due to the failure. And then, the scheduler selects a new processor for the paused tasks (lines 25-29). Once the schedule is determined, the scheduler notifies the destination processor to fetch the required data from the master. In this algorithm, we use $rest$ and $reft$ to denote the reschedule earliest start and finish time, respectively. Assumed that the failure time of processor p_k is T_f, for the entry tasks, the $rest$ is calculated as

$$rest(v_{entry}, p_k) = \max\{T_f, avail(p_k)\}, \tag{14}$$

For the other tasks, their required data must be fetched from the master, so its $rest$ is calculated as

$$rest(v_i, p_k) = \max\{T_f + \sum_{v_j \in Pare(v_i)} c(e_{ji}), avail(p_k)\}. \tag{15}$$

The key of FTDR is the failure detecting. In our paper, the supervisor keeps listening the "heartbeat" signals which are sent from each processor and analyzes the states of them. Hence, the resource overhead of the supervisor is considerable. Whereas, this part of overhead does not affect the function of the scheduler and the performance of application execution. Moreover, the processors send "heartbeat" signals to the supervisor, which consumes a small part of resources. However, compared with the resource overhead generated by the existing algorithms adopting the replication strategy, e.g. MaxRe and FTSA, the resource overhead generated by sending signals is much smaller and negligible.

Of course, there are many other possible events. For example, the application properly executes but does not produce any results (files). This kind of events can be easily handled without consuming much time, and the response to these events such as the successful writing to files should be a part of work included in each subtask.

5.4 Illustrated Examples

In Fig. 4, we give two examples to illustrate the FTDR algorithm. Figure 4a and b show the schedules generated by FTDR when a fault occurs at time 4 and 7 whose recovery time is 5 and 9, respectively. In Fig. 4a, the schedule of the entry task v_0 is generated first and it is allocated to processor p_1. During its execution, a fault occurs at time 4. When the fault is checked by the processor supervisor, the supervisor notices the scheduler to reschedule v_0 and its new allocated processor is p_0. After v_0 is completed at time 10, the two new ready tasks v_1 and v_2 are scheduled, and they are allocated to p_0 and p_2, respectively. During

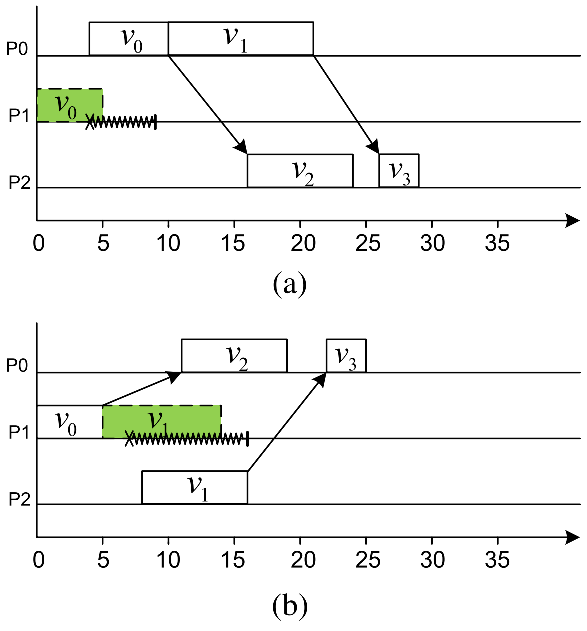


Fig. 4 **a** A schedule generated by FTDR when a failure occurs at time 4 and **b** A schedule generated by FTDR when a failure occurs at time 7

their execution, no fault occurs. When v_1 and v_2 are all finished, v_3 is allocated to p_2 and completed successfully at time 29. Figure 4b gives another schedule when a fault occurs at time 7. When v_0 completes its execution, tasks v_1 and v_2 become ready and their allocated processors are p_1 and p_2, respectively. When v_1 is executing at p_1, it is paused due to a fault at time 7. Hence, v_1 is reallocated to p_2 and the rest tasks are executed successfully. From above examples we can see that the FTDR algorithm can schedule application flexibly under arbitrary faults.

5.5 Complexity and Overhead

The time complexity of FTDR is expressed in terms of the number of nodes $|V|$, the number of edges $|E|$, the number of processors $|P|$, the number of failures ε, and the maximum in degree and out degree of tasks D_{in} and D_{out}.

First, task priority is calculated for each task. Because the calculation of a task's priority relies on the priority values of all its children, its complexity is $O(D_{out})$. Since the summation of the D_{out} of all nodes is $\sum D_{out} = |E|$, the total complexity of task priority calculating phase is $O(|E|)$. In priority queue generating phase, quick sort algorithm is adopted and the complexity is $O(|V|\log|V|)$. Hence, the total complexity is $O(|E| + |V|\log|V|)$.

When an application is submitted or its subtasks are completed, the scheduler responds to application submission event or task completion event to generate the schedule of new ready tasks. The est and eft values of each task on all processors are calculated and the calculation of est and eft relies on the dat of all parents. Hence, the complexity is $O(|P||V|D_{in})$. In addition, each generated schedule is inserted into the schedule queue in order and the complexity is $O(|V|^2)$.

The scheduler responds to task start event for $|V|$ times. At each task start event, the first task is dispatched to its allocated processor. It is removed from the ready queue, and its schedule is removed from the schedule queue. The complexity of this stage is $O(|V|)$.

Due to the failures, some tasks need to be rescheduled. Assumed the number of failures is ε. At each processor failure event, all tasks (at most $|V|$) assigned to the failure processor must be rescheduled, the complexity is $O(\varepsilon|v||P|)$. At each processor recovery event, the information of the available processors is updated. The complexity is $O(1)$.

In conclusion, the total algorithm complexity is $O(|P||V|(D_{in} + \varepsilon) + |V|^2)$.

6 Experimental Results and Analysis

To evaluate the performance of the proposed algorithm FTDR, we conduct series of simulations. In this section, we first present experiment parameters and performance metrics. And then, experimental results are presented and we give detailed analysis for each figure.

6.1 Experiment Parameters

In our experiments, we use randomly generated graphes, whose parameters are consistent with those used in [2, 7, 43]. The generated parameters of the random graphs are listed as follows.

- DAG size, $|V|$: The number of tasks in the application DAG.
- Communication to computation cost ratio, CCR: The average communication cost divided by the average computation cost of the application DAG.
- Parallelism factor, λ: The number of levels of the application DAG is generated randomly, using a uniform distribution with a mean value of $\frac{\sqrt{|V|}}{\lambda}$, and then rounding it up to the nearest integer. The width is generated using a uniform distribution with a mean value of $\lambda\sqrt{|V|}$, and then rounding it up to the nearest integer. A low λ leads to a DAG with a low parallelism degree [7].

In each experiment, the values of these parameters are assigned from the corresponding sets given below. A parameter should be assigned by all values given in its set in a single experiment.

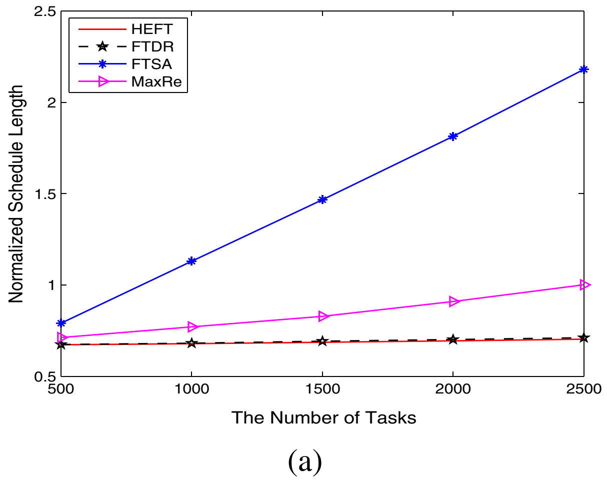
- $SET_V = \{500, 1000, 1500, 2000, 2500\}$
- $SET_{CCR} = \{0.2, 0.5, 1, 2, 5\}$
- $SET_\lambda = \{0.2, 0.5, 1, 2, 5\}$

To generate a DAG with a given number of tasks, parallelism factor λ and CCR, the number of levels is determined by λ firstly, and then the number of tasks at each level is determined. Edges are only generated between the nodes in the adjacent levels, obeying 0-1 distribution. The computation time of each task is selected randomly from an uniform distribution with range $[10, 50]$. To obtain the desired CCR for a graph, the communication time of each task is also randomly selected from an uniform distribution, whose mean depends on the product of CCR value and the average computation time.

Moreover, the number of processors and the failure percentage are two important factors which be discussed in our experiments. The two parameters are described as follows.

- System size, $|P|$: The number of processors in the heterogenous computing system, and the set is {8,16,32,64,128}.
- Failure ratio, Γ: The expected number of occurrence of failures in unit of time, which are {0.00001,0.00002,0.00003,0.00004,0.00005}.

Combining with the DAG types, there are 2500 experiment cases, among which we select 100 cases. For each case, 50 random DAGs are generated and tested to avoid scattering effects. The results are averaged over the 50 tested values for each case. Experiments based on diverse DAG types prevent biasing toward a particular scheduling algorithm.



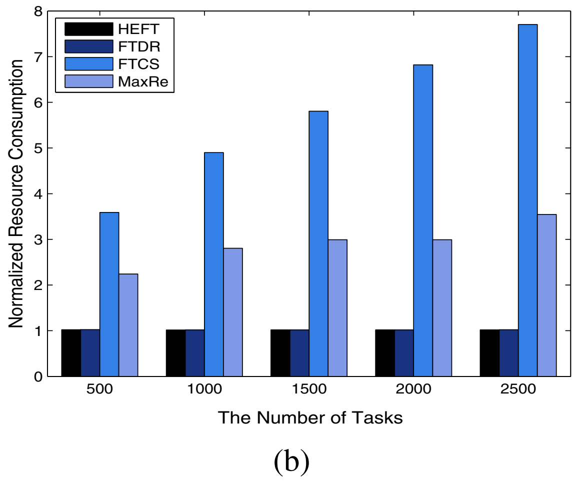


Fig. 5 **a** Average NSL and **b** average NRC versus graph size for different algorithms ($|P|$=64, CCR=1.0, λ=1.0, $\Gamma = 0.00003$)

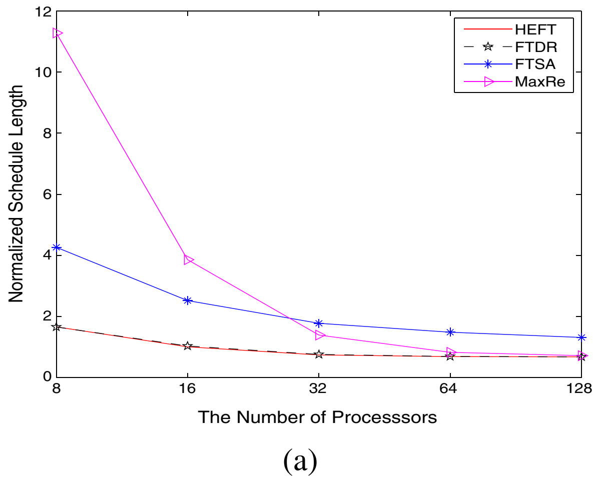
6.2 Performance Metrics

The proposed algorithm FTDR is compared with three other scheduling algorithm, including HEFT [2], FTSA [22], and MaxRe [23], respectively. We have modified the three algorithms to adapt the scheduling problem discussed in this paper. The comparisons of the algorithms are based on the following three metrics:

- **Normalized Schedule Length (NSL).** The main performance measure of a scheduling algorithm on a graph is the schedule length (makespan) of its output schedule. Since a large set of task graphs with different properties are used, it is necessary to normalize the schedule length to a bound, which is called the *Normalized Schedule Length (NSL)*. The NSL value of an algorithm on a graph is defined by

$$NSL = \frac{makespan}{the\ length\ of\ CP_{max}} \tag{16}$$

For an unscheduled DAG, if the computation cost of each task v_i is set with the maximum value, then the critical path will be based on maximum computation costs, which is represented as



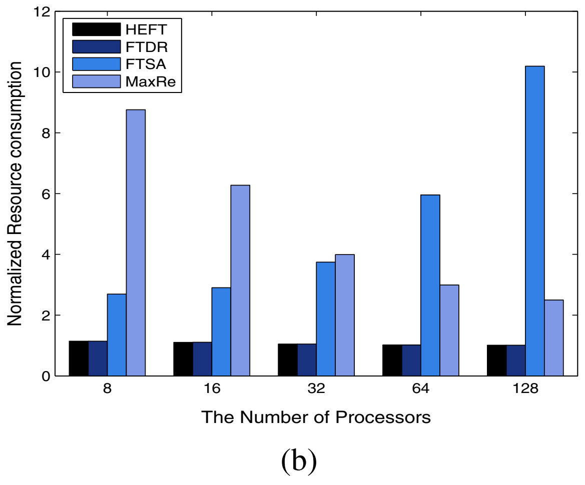


Fig. 6 **a** Average NSL and **b** average NRC versus system size for different algorithms ($|V|$=1500, CCR=1.0, λ=1.0, $\Gamma = 0.00003$)

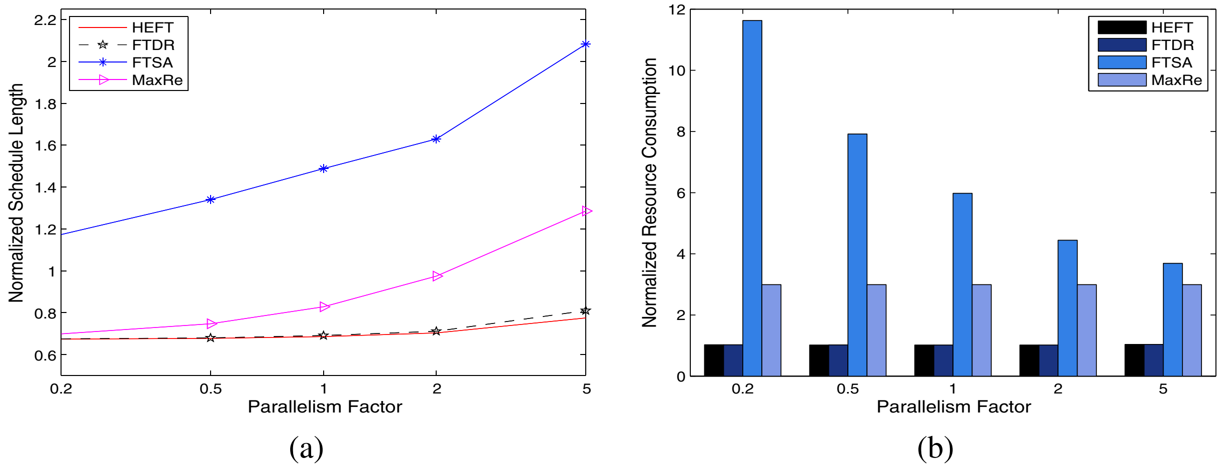


Fig. 7 **a** Average NSL and **b** average NRC versus parallelism factor for different algorithms ($|P|$=64, $|V|$=1500, CCR=1.0, $\Gamma = 0.00003$)

CP_{max}. The denominator is the length of CP_{max}, including the computation costs and the commination costs. The NSL of a graph can be less than one when the graph width is less than the number of processors, also, it can be greater than one when the graph width is greater than the number of processors. Average NSL values over several task graphs are used in our experiments.

- **Normalized Resource Consumption (NRC).** The resource consumption is another metric to measure the performance of scheduling algorithms since the compared algorithms, except HEFT, aim at achieving a high success rate by using the active replication strategy. Similarly, it is necessary to normalize the resource consumption to a bound, which is called the *Normalized Resource Consumption (NRC)*. The NRC value of an algorithm on a graph is defined by

$$NRC = \frac{resource\ consumption}{\sum_{v_i \in V} min_{p_j \in P}\{w_{i,j}\}} \tag{17}$$

The denominator is the summation of the minimum resource consumption of all tasks. Average

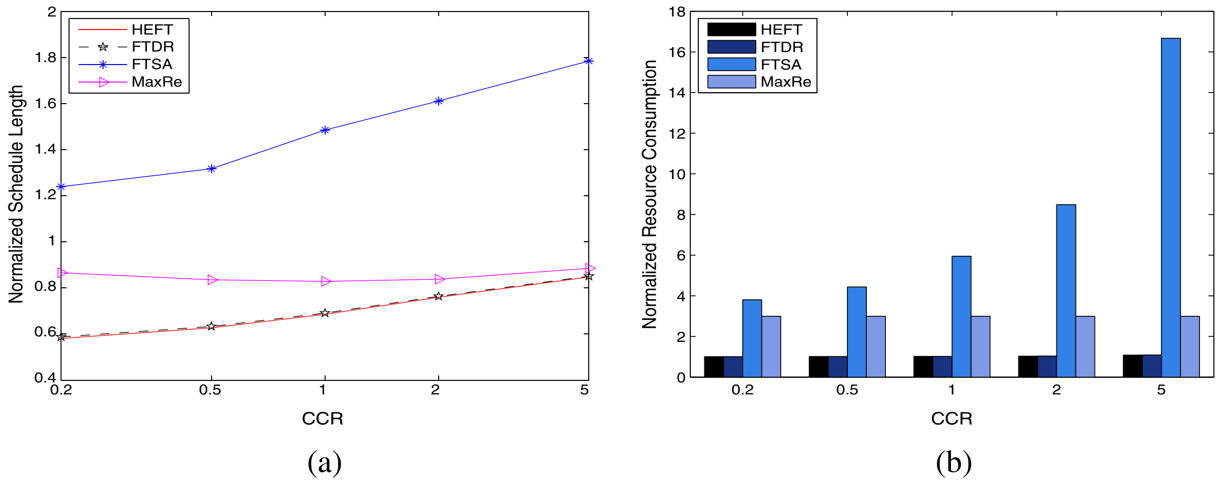


Fig. 8 **a** Average NSL and **b** average NRC versus CCR for different algorithms ($|P|$=64, $|V|$=1500, λ=1.0, $\Gamma = 0.00003$)

Table 4 Failure rate versus the number of tasks ($|P|$=64, CCR=1.0, λ=1.0, $\Gamma = 0.00003$)

$\|V\|$	500	1000	1500	2000	2500
HEFT	34.8 %	62.2‘%	72 %	83 %	87.2 %
FTDR	0 %	0 %	0 %	0 %	0 %
FTSA	25.4 %	22 %	17.4 %	16 %	14.6 %
MaxRe	48.4 %	59.6 %	63 %	66.8 %	69.4 %

NRC values over several task graphs are used in our experiments.

- **Application Failure Percentage (AFP).** The application failure percentage is to measure the probability that an application fails when it is scheduled using different algorithms. In our paper, it is defined as the ratio of the times that applications fail to the total times.

6.3 A Brief Description of the Compared Algorithms

In order to compare our algorithm to HEFT, FTSA, and MaxRe, we give here a brief description of those algorithms, using the original notations of [2, 22] and [23], respectively.

- HEFT (Heterogeneous Earliest-Finish-Time) [2] is a classical static list scheduling algorithm. It does not take the processor failure into consideration. However, it is a good refer to test the performance of FTDR.
- FTSA (Fault-Tolerant Scheduling Algorithm) [22] is an extended version of the classic HEFT algorithm. It allocates $\varepsilon + 1$ copies of each task to different processors to tolerate an arbitrary number ε of fail-silent processor failures.
- MaxRe [23] incorporates the reliability analysis into the active replication scheme, and exploits a dynamic number of replicas for different tasks. Given the processor reliability, MaxRe calculates different number of replicas for different tasks to meet the user's reliability requirement. In our algorithm, the processor reliability is set as {0.99999,0.99998,0.99997,0.99996,0.99995} according to (2) and the failure ratio, and user's reliability requirement is 0.9999.

6.4 Performance Results

The results are organized in three parts. The first part presents results on makespan and resource consumption. The second part presents results on the execution failure rate. The last part presents results with respect to the processor failure percentage. A detailed analysis is given for each set of experiments.

6.4.1 Makespan and Resource Consumption

The performance of the four algorithms are compared with respect to various graph characteristics. In the first set of experiments, we compare the performance in terms of makespan and resource consumption of the algorithms with respect to various graph size (see Fig. 5). From Fig. 5a we can see that, the NSL-based performance ranking of the algorithms is {HEFT, FTDR, FTSA, MaxRe}. (Each ranking in this paper starts with the best algorithm and ends with the worst one with respect to the given comparison metric.) The HEFT

Table 5 Failure rate versus the number of processors ($|V|$=1500, CCR=1.0, λ=1.0, $\Gamma = 0.00003$)

$\|P\|$	8	16	32	64	128
HEFT	67.4 %	66.8 %	74.2 %	70.8 %	76 %
FTDR	0 %	0 %	0 %	0 %	0 %
FTSA	11.8 %	15.2 %	19.6 %	21.8 %	17.6 %
MaxRe	0.6 %	8.8 %	29.2 %	63.2 %	81.8 %

Table 6 Failure rate versus λ ($|P|$=64, $|V|$=1500, CCR=1.0, $\Gamma = 1.0$)

λ	0.2	0.5	1	2	5
HEFT	73.6 %	72.4 %	73 %	70 %	76.2 %
FTDR	0 %	0 %	0 %	0 %	0 %
FTSA	23 %	21.6 %	20.6 %	15.8 %	9.8 %
MaxRe	88.2 %	76.2 %	62.6 %	50.6 %	39.2 %

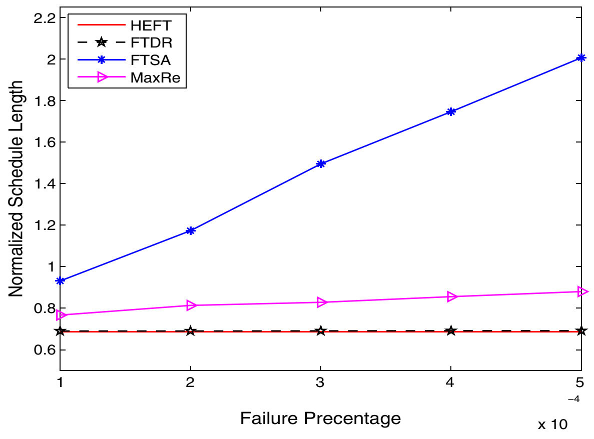
algorithm achieves the lowest makespan value. The makespan value of the proposed algorithm FTDR is very close to that of HEFT and FTDR only lose about 1.44 % performance in terms of makespan to guarantee the success of applications execution. In contrast, both of FTSA and MaxRe sacrifice a large amount of makespan compared with HEFT and FTDR, aiming at achieving a higher reliability by allocating multiple copies of each task to different processors. The experimental results show that the average NSL values of FTSA and MaxRe are worse than FTDR by 161.03 % and 240.65 %, respectively. Increasing the number of tasks affects the NSL values of HEFT and FTDR slightly, but greatly for FTSA, this is because an increasing number of tasks lead to a significant increasing of replicas, hence leading to a more serious latency. The figure shows that the performance of MaxRe falls in between FTSA and the other two, because the number of replicas generated by MaxRe is much less than FTSA but more than HEFT and FTDR. The varying trend of makespan is same with that of resource consumption.

Figure 5b gives the comparison of the NRC-based performance of the algorithms, and their ranking on resource consumption is {HEFT, FTDR, FTSA, MaxRe} which is consistent with the NSL-based performance. FTDR only reschedules those tasks that are suspended due to failure, which do not lead to much extra resource consumption. However, FTSA and MaxRe, allocate multiple copies of each task to different processors no matter that the failures affect its execution, which leads to a large amount of resource overheads. Increasing the number of tasks affect the NRC value of FTSA and MaxRe, and the reasons for the two algorithms are different. For FTSA, more tasks leads to longer makespan, hence, more failures occur during the execution, which leads to more replicas as well as resource consumption for each task. While for MaxRe, more tasks lead to longer makespan as well and the reliability of a task replica decreases with the increasing execution time. In order to satisfy the demand reliability, more replicas are generated. Hence the resource consumption increases with the increasing number of tasks.

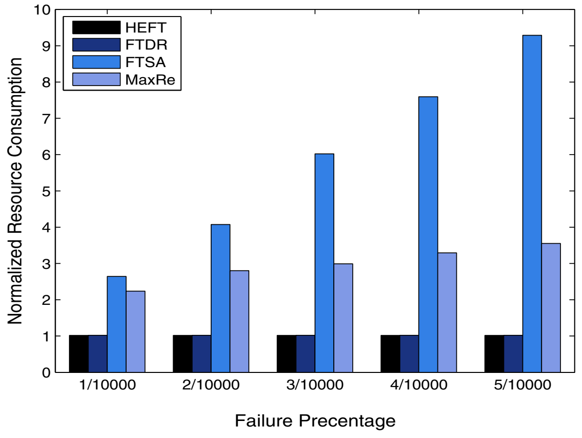
In the second set of experiments, we explore the relationship between the system size and the performance of the algorithms. The experimental results in Fig. 6 show that, the performance of HEFT and FTDR are very close to each other. As a fault-tolerant scheduling algorithm, FTDR performs much better than the compared algorithms FTSA and MaxRe not only in makespan but also in resource consumption. Along with the increasing number of processors, the degree of parallelism gets higher, so the NSL values of four algorithms descend. However, MaxRe shows a quicker downward trend than the other three algorithms. When the number of processors in the system is less than 16, the NSL value of MaxRe is greater than that of FTSA, while when the number of processors gets greater, it becomes smaller than that of FTSA. That is because the number of replicas of each task is determined by the number of failures using FTSA but

Table 7 Failure rate versus CCR ($|P|$=64, $|V|$=1500, λ =1.0, $\Gamma = 0.00003$)

CCR	0.2	0.5	1	2	5
HEFT	70.6 %	71.6 %	72.4 %	73.4 %	74.4 %
FTDR	0 %	0 %	0 %	0 %	0 %
FTSA	19.8 %	21.2 %	19.6 %	15.2 %	14 %
MaxRe	45.6 %	51.6 %	66.2 %	77 %	92.6 %



(a) Average NSL versus failure percentage for different algorithms (|P|=64, |V|=1500, CCR=1.0, λ=1.0)



(b) Average NRC versus failure percentage for different algorithms (|P|=64, |V|=1500, CCR=1.0, λ=1.0)

Fig. 9 Performance comparison among different algorithms

the reliability using MaxRe. Hence, FTSA schedules each task for less times than MaxRe when the number of processors is small, and more times with more processors. From Fig 6b, the NRC values of two algorithms show the opposite trends with the increasing number of processors.

The NRC value of MaxRe is much higher than that of FTSA firstly, and than it declines until the NRC value of MaxRe is lower than that of FTSA. That is because the number of replicas using MaxRe is increasing with the increasing makespan. Using FTSA, the number of processors is doubled but the makespan is shortened slightly, which leads to a great increase of execution time as well as failure times. Hence, more replicas are needed for each tasks with the increasing number of processors, and the NRC value of FTSA shows an upward trend.

The next set of experiments are with respect to the graph structure. In total, the experimental results in Fig. 7 show that the performance ranking in terms of makespan and resource consumption of the four algorithms is {HEFT, FTDR, MaxRe,FTSA}. The NSL values show an upward trend while the NRC values show a downward or steady trend with the increasing λ. The reasons are explained as follows. When λ is small, the generated graph is with a low degree of parallelism, so the length of the critical path is long. Along with the increasing λ, the length of the critical path of the generated graph becomes shorter. Although the makespan is reduced, the NSL values of the algorithms still show an upward trend. The NRC value of FTSA is reduced with the the increasing λ, as the reduced makespan leads to less failures.

In the fourth set of experiments, the performance with respect to CCR is compared (see Fig. 8). The performance ranking in terms of makespan and resource consumption of the algorithms is still {HEFT, FTDR, MaxRe, FTSA}. From Fig. 8a we can see that, the NSL values of three algorithms except MaxRe all show upward trends along with the increasing CCR

Table 8 Execution failure rate versus processor failure percentage (|P|=64, |V|=1500, λ =1.0, CCR=1.0)

Failure Percentage	0.00001	0.00002	0.00003	0.00004	0.00005
HEFT	36.2 %	57.6 %	72.2 %	83.6 %	87.8
FTDR	0 %	0 %	0 %	0 %	0 %
FTSA	17.2 %	18.6 %	19.6 %	19 %	18 %
MaxRe	38.4 %	48.8 %	62 %	73.8 %	74.2 %

value. The explanation is as follows. When CCR is small, the computation cost of applications dominates the communication cost. The communication cost is low, so the makespan of the generated schedules is small. When CCR is great, parallelization leads to a great amount of time overhead, which increases the makespan. The NSL values of MaxRe keep steady because the makespan is increasing linearly with the length of critical path. Figure 8b shows that the NRC value of FTSA increases sharply with the increasing CCR but the other three not. That is because the number of replicas using FTSA depends on the number of failures, which is greater with a greater makespan. While the number of a task's replicas using MaxRe depends on the total execution time before its execution. Although the makespan is increasing, the total execution time won't change, hence leading to an unchange number of replicas.

6.4.2 Execution Failure Rate

Besides above two performance metrics, application failure percentage (AFP) is compared to measure the quality of schedules generated by the algorithms. The statistical data is given in Tables 4–7, respectively.

From the four tables we can see that, the AFP value of HEFT is highest in average, as HEFT is a scheduling algorithm without considering the processor failures. The algorithm with the second highest AFP value is MaxRe, followed by FTSA. That is because the number of replicas of each task using MaxRe is less than that using FTSA. The analysis indicates that for static scheduling algorithms, the more the number of replicas is, the higher the success rate is. Table 5 shows that the two fault-tolerant scheduling algorithms, FTSA and MaxRe, cannot tolerate all processor failures. In contrast, the proposed algorithm FTDR can tolerate all processor failures, because it is a dynamic scheduling algorithm, and it can deal with failures in time once they occur.

6.4.3 Application Failure Percentage

In above experiments, the processor failure percentage is set as 0.00003. Next, we explore how the failure ratio affects the performance of the algorithms. Figure 9 presents the performance in terms of makespan and resource consumption of the four algorithms. According to Fig. 9, the processor failure ratio does not affect the makespan and resource consumption of HEFT, FTDR, and MaxRe greatly. The reasons are explained respectively as follows. HEFT is an algorithm without taking failure into consideration, hence it is not affected by the processor failure ratio. FTDR is to reschedule tasks only when failures occur, which leads to a small time overhead, hence the performance deterioration is slight. MaxRe determines the number of replicas of each task based on user's required reliability and the processor reliability. The change of failure ratio affects slightly the processor reliability, so the number of replicas does not increase. In contract, because the number of replicas of FTSA is determined by the number of failures, which is increasing linearly with the increasing failure ratio, the makespan and resource consumption of the schedules generated by it grow quickly.

Table 8 presents the application failure percentage of the algorithms with respect to processor failure ratio. The HEFT algorithm gives the worst performance. The application failure percentage of FTDR is zero since it is a dynamic rescheduling algorithm. FTSA and MaxRe show a better performance than HEFT, and FTSA outperforms MaxRe because the number of replicas using FTSA is more than that using MaxRe.

7 Conclusions

This paper presents a novel fault-tolerate dynamic rescheduling algorithm for heterogenous computing systems, called FTDR. It adopts a reschedule strategy which is different from the active replication scheme, and overcomes its drawbacks. FTDR keeps listening to five kinds of events, including application submitted event, task start event, task completed event, processor failure event and processor recovery event. Once a processor failure is detected, the scheduler reschedules the tasks that are located on the failure processor, and reassigns them to the objective processors. Hence, it does not waste much resource, but can tolerate any number of processor failures. According to the experimental results, we have shown that FTDR is superior to FTSA and MaxRe both in terms of resource consumption and makespan. We also point out that the schedules generated by

FTDR can tolerate fault no matter how many failures occur.

In this paper, we assume that the computation and communication costs are known in advance and the knowledge is accurate. However, in general, the priori information is not always accurate, which affects the performance of the scheduling problem and even the proposed algorithm is not suitable for scheduling the application with inaccurate knowledge. Hence, in the future, we will further the research of this situation.

Acknowledgments The authors thank the anonymous reviewers for their valuable comments and suggestions. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61173013, 61202109, and 61472126).

References

1. Kasahara, H., Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans. Comput. **33**(11), 1023–1029 (1984)
2. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002)
3. Daoud, M.I., Kharma, N.: A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **68**(4), 399–409 (2008)
4. Nesmachnow, S., Dorronsoro, B., Pecero, J., Bouvry, P.: Energy-aware scheduling on multicore heterogeneous grid computing systems. J. Grid Comput. **11**(4), 653–680 (2013)
5. Arabnejad, H., Barbosa, J.: A budget constrained scheduling algorithm for workflow applications. J. Grid Comput. **12**(4), 665–679 (2014)
6. Ranaweera, S., Agrawal, D.: A scalable task duplication based scheduling algorithm for heterogeneous systems. In: Proceedings of 2000 International Conference on Parallel Processing, pp. 383–390 (2000)
7. Bansal, S., Kumar, P., Singh, K.: An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. IEEE Trans. Parallel Distrib. Syst. **14**(6), 533–544 (2003)
8. Shin, K., Cha, M., Jang, M., Jung, J., Yoon, W., Choi, S.: Task scheduling algorithm using minimized duplications in homogeneous systems. J. Parallel Distrib. Comput. **68**(8), 1146–1156 (2008)
9. Tang, X., Li, K., Liao, G., Li, R.: List scheduling with duplication for heterogeneous computing systems. J. Parallel Distrib. Comput. **70**(4), 323–329 (2010)
10. Song, I., Yoon, W., Jang, E., Choi, S.: Task scheduling algorithm with minimal redundant duplications in homogeneous multiprocessor system in Grid and Distributed Computing, pp. 238–245. Springer (2011)
11. Bansal, S., Kumar, P., Singh, K.: An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. IEEE Trans. Parallel Distrib. Syst. **14**(6), 533–544 (2003)
12. Hagras, T., brevecek, J.J.: A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. Parallel Comput. **31**(7), 653–670 (2005)
13. Liou, J., Palis, M.: An efficient task clustering heuristic for scheduling dags on multiprocessors. In: Proceedings of Parallel and Distributed Processing Symposium (1996)
14. Fangfa, F., Yuxin, B., Xinaan, H., Jinxiang, W., Minyan, Y., Jia, Z.: An objective-flexible clustering algorithm for task mapping and scheduling on cluster-based noc. In: 2010 10th Russian-Chinese Symposium on Laser Physics and Laser Technologies (RCSLPLT) and 2010 Academic Symposium on Optoelectronics Technology (ASOT), 28 2010-aug. 1 2010, pp. 369 –373
15. Khan, M.A.: Scheduling for heterogeneous systems using constrained critical paths. Parallel Comput. **38**(4), 175–193 (2012)
16. Stearley, J.: Defining and measuring supercomputer reliability, availability, and serviceability (ras). In: Proceedings of the Linux Clusters Institute Conference (2005)
17. Rahman, R.M., Barker, K., Alhajj, R.: Replica placement strategies in data grid. J. Grid Comput. **6**(1), 103–123 (2008)
18. Yang, H., Luan, Z., Li, W., Qian, D.: Mapreduce workload modeling with statistical approach. J. grid Comput. **10**(2), 279–310 (2012)
19. Koo, R., Toueg, S.: Checkpointing and rollback-recovery for distributed systems. IEEE Trans. Softw. Eng. **1**, 23–31 (1987)
20. Chakravorty, S.: A fault tolerance protocol for fast recovery. ProQuest (2008)
21. Yang, X., Wang, Z., Xue, J., Zhou, Y.: The reliability wall for exascale supercomputing. IEEE Trans. Comput. **61**(6), 767–779 (2012)
22. Benoit, A., Hakem, M., Robert, Y.: Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In: IEEE International Symposium Parallel Distributed Processing, pp. 1–8. IEEE (2008)
23. Zhao, L., Ren, Y., Xiang, Y., Sakurai, K.: Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems. In: 12th IEEE International Conference High Performance Computing Communications, pp. 434–441. IEEE (2010)
24. Shatz, S.M., Wang, J.-P., Goto, M.: Task allocation for maximizing reliability of distributed computer systems. IEEE Trans. Comput. **41**(9), 1156–1168 (1992)
25. Qin, X., Jiang, H.: A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. J. Parallel Distrib. Comput. **65**(8), 885–900 (2005)

26. Dongarra, J.J., Jeannot, E., Saule, E., Shi, Z.: Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pp. 280–288. ACM (2007)
27. Jeannot, E., Saule, E., Trystram, D.: Bi-objective approximation scheme for makespan and reliability optimization on uniform parallel machines. In: Euro-Par 2008–Parallel Processing, pp. 877–886. Springer (2008)
28. Girault, A., Saule, E., Trystram, D.: Reliability versus performance for critical applications. J. Parallel Distrib. Comput. **69**(3), 326–336 (2009)
29. Tang, X., Li, K., Li, R., Veeravalli, B.: Reliability-aware scheduling strategy for heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **70**(9), 941–952 (2010)
30. Boeres, C., Sardiña, I.M., Drummond, L.: An efficient weighted bi-objective scheduling algorithm for heterogeneous systems. Parallel Comput. **37**(8), 349–364 (2011)
31. Jeannot, E., Saule, E., Trystram, D.: Optimizing performance and reliability on heterogeneous parallel systems: Approximation algorithms and heuristics. J. Parallel Distrib. Comput. **72**(2), 268–280 (2012)
32. Tao, Y., Jin, H., Wu, S., Shi, X., Shi, L.: Dependable grid workflow scheduling based on resource availability. J. Grid Comput. **11**(1), 47–61 (2013)
33. Hakem, M., Butelle, F.: Reliability and scheduling on systems subject to failures. In: International Conference on Parallel Processing, pp. 38–38. IEEE (2007)
34. Qin, X., Jiang, H.: A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. Parallel Comput. **32**(5), 331–356 (2006)
35. Zheng, Q., Veeravalli, B.: On the design of communication-aware fault-tolerant scheduling algorithms for precedence constrained tasks in grid computing systems with dedicated communication devices. J. Parallel Distrib. Comput. **69**(3), 282–294 (2009)
36. Zheng, Q., Veeravalli, B., Tham, C.-K.: On the design of fault-tolerant scheduling strategies using primary-backup approach for computational grids with low replication costs. IEEE Trans. Comput. **58**(3), 380–393 (2009)
37. Benoit, A., Hakem, M., Robert, Y.: Realistic models and efficient algorithms for fault tolerant scheduling on heterogeneous platforms. In: 37th International Conference on Parallel Processing, pp. 246–253. IEEE (2008)
38. Khokhar, A., Prasanna, V., Shaaban, M., Wang, C.-L.: Heterogeneous computing: challenges and opportunities. Computer **26**(6), 18–27 (1993)
39. Radulescu, A., Van Gemund, A.: Fast and effective task scheduling in heterogeneous systems. In: Proceedings of 9th Heterogeneous Computing Workshop, pp. 229–238 (2000)
40. Choudhury, P., Chakrabarti, P., Kumar, R.: Online scheduling of dynamic task graphs with communication and contention for multiprocessors, vol. 23, pp. 126–133 (2012)
41. Young, J.W.: A first order approximation to the optimum checkpoint interval. Commun. ACM **17**(9), 530–531
42. Jin, H., Sun, X.-H., Zheng, Z., Lan, Z., Xie, B.: Performance under failures of dag-based parallel computing. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 236–243 (2009)
43. Daoud, M.I., Kharma, N.: A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **68**(4), 399–409 (2008)