Hindawi
Wireless Communications and Mobile Computing
Volume 2019, Article ID 4807502, 12 pages
https://doi.org/10.1155/2019/4807502

Research Article

A Selective Mirrored Task Based Fault Tolerance Mechanism for Big Data Application Using Cloud

Hao Wu[1], **Qinggeng Jin**[2], **Chenghua Zhang**[1], **and He Guo**[1]

[1]*School of Software Technology, Dalian University of Technology, Dalian, Liaoning, China*
[2]*Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, Nanning, Guangxi, China*

Correspondence should be addressed to Qinggeng Jin; jinqinggeng@aliyun.com

Received 6 December 2018; Accepted 29 January 2019; Published 26 February 2019

Guest Editor: Salimur Choudhury

With the wide deployment of cloud computing in big data processing and the growing scale of big data application, managing reliability of resources becomes a critical issue. Unfortunately, due to the highly intricate directed-acyclic-graph (DAG) based application and the flexible usage of processors (virtual machines) in cloud platform, the existing fault tolerant approaches are inefficient to strike a balance between the parallelism and the topology of the DAG-based application while using the processors, which causes a longer makespan for an application and consumes more processor time (computation cost). To address these issues, this paper presents a novel fault tolerant framework named Fault Tolerance Algorithm using Selective Mirrored Tasks Method (FAUSIT) for the fault tolerance of running a big data application on cloud. First, we provide comprehensive theoretical analyses on how to improve the performance of fault tolerance for running a single task on a processor. Second, considering the balance between the parallelism and the topology of an application, we present a selective mirrored task method. Finally, by employing the selective mirrored task method, the FAUSIT is designed to improve the fault tolerance for DAG based application and incorporates two important objects: minimizing the makespan and the computation cost. Our solution approach is evaluated through rigorous performance evaluation study using real-word workflows, and the results show that the proposed FAUSIT approach outperforms existing algorithms in terms of makespan and computation cost.

1. Introduction

Recent years have witnessed that the big data analysis grows dramatically, and the related applications have been used everywhere in both academia [1] and industry [2]. There is no denying that the developmental cloud platform technologies played a key role in this process; the plenty of processers in cloud make sure that the scholars can handle the significant large-scale big data processing [3–6]. However, due to voltage fluctuation, cosmic rays, thermal changes, or variability in manufacturing, the chip level soft errors and the physical flaws are inevitable for a processer even when the probability of that is extremely low [7, 8]. Furthermore, the abundant use of processers by a big data application induces that the probability cannot be ignored.

Big data and big data analysis have been proposed for describing data sets as analytical technologies in large-scale complex programs, which need to be analyzed with advanced analytical methods [9, 10]. No matter whether the big data applications are developed for commercial purposes or scientific researches, most of these applications require significant amount of computing resources, such as market structure analysis, customer trade analysis, environmental research, and astrophysics data processing. Motivated by the reasonable price, rapid elasticity, and shifting responsibility of maintenance, backups, and management to cloud providers, more and more big data applications have been deployed to clouds, such as EC2 [11], Google Cloud [12], and Microsoft Azure [13].

The clouds provide unlimited computing resources (from the user's point of view) including CPU resources and GPU resources. The on-demand recourses facilitate users to choose apposite processors (with CPU or GPU resources) for executing their big data applications efficiently [14]. However,

according to [15], there are many factors such as voltage fluctuation, cosmic rays, thermal changes, or variability in manufacturing, which cause the processors (both CPU and GPU) to be more vulnerable. Indeed, the probability of fault rate is really low. But, as already noted, plenty of processors participate in computing the big data application, and the computing time of each processor may be very long. These potential factors will lead to an exponential growth for the fault rate in a cycle of running a big data application.

The failures caused by processors are disastrous for a big data application which is deployed on the processors; i.e., once a fault occurs on any processor, the application will have to be executed all over again, and that will waste lots of monetary cost and time cost. Thus, improving the robustness (or reducing the fault rate) for running a big data application has attracted many scholars' attention; many studies have been exploring this problem. According to the literature, these studies are classified into two main categories: resolving the problem in hardware or software level.

From the hardware level's perspective, improving the mean time between failures (MTBF) [16] of the processors is the key to reduce the fault rate for running a big data application. As everyone knows, it is impossible to eradicate the failures in a processor. Apart from lifting tape-out technology, the [15] proposed an adaptive low-overhead fault tolerance mechanism for many-core processor, which treats fault tolerance as a device that can be configured and used by application when high reliability is needed. Although [15] improves the MTBF, but the risk of occurring failures remains. Therefore, the other scholars seek solutions in software level.

From the software level's perspective, the developed check-point technology [17] makes sure that a big data application can be completed under any size of MTBF. Thus, many check-point strategies are proposed to resolve this problem such as [18, 19]. These strategies only pay a little extra cost, but they can complete the applications under any size of MTBF. As a result, almost all of cloud platforms provide check-point interface for users. However, these strategies did not consider the date dependencies between the processors, which make it inappropriate to big data application running on cloud.

In order to handle the DAG based applications, the copy task based method (also known as primary backup based method) is proposed to resolve the problem. In [20], Qin and Jiang proposed an algorithm eFRD to enable a systems fault tolerance and maximize its reliability. On the basis of eFRD, Zhu et al. [21] developed an approach for task allocation and message transmission to ensure faults can be tolerated during the workflow execution. But the task based methods will make the makespan have a long delay due to the fact that the backup tasks are not starting with the original tasks.

To the best of our knowledge, for the big data application, there are no proper check-point strategies which can handle the data dependencies among the processors simultaneously. In this paper, we propose a novel check-point strategy for big data applications, which considers the effect of the data communications. The proposed strategy adopts high level failure model to resolve the problem, which makes it closer to practice. Meanwhile, the subsequent effect caused by data dependencies after a failure is also considered in our strategy.

The main contributions of this paper are as follows:

(i) A selective mirrored task method is proposed for the fault tolerance of the key subtasks in a DAG based application.

(ii) On the basis of the selective mirrored task method, a novel check-point framework is proposed to resolve the fault tolerance for a big data application running on cloud; the framework is named Fault Tolerance Algorithm using Selective Mirrored Tasks Method (FAUSIT).

(iii) A thorough performance analysis is conducted for FAUSIT through experiments on randomly generated test big data application as well as real-world application traces.

The rest of this paper is structured as follows. The related work is summarized in Section 2. Section 3 introduces the models of the big data application, the cloud platform, and the MTBF and then formally defines the problem the paper is addressing. Section 4 presents the novel check-point strategy for the big data application running on cloud. Section 5 conducts extensive experiments to evaluate the performance of our algorithm. Section 6 concludes the paper with summary and future directions.

2. Related Work

Over the last two decades, owing to the increasing scale of the big data applications [22], the fault tolerance for big data application is becoming more and more crucial. Considerable research has been explored by scholars. In this section, we summarize the research in terms of theoretic methods and heuristic methods.

A number of theoretic methods have been explored by scholars. For a task running on a processor, Young [21] proposed a first order failure model and figured out an approximation to the optimum check-point interval which is $\varphi_{opt} = \sqrt{2\delta M}$, where δ is the time to write a check-point file, M is the mean time between failures for the processor, and φ_{opt} is the optimum computation interval between writing check-point files. However, the model in Young [21] will never have more than a single failure in any given computation interval. This assumption goes against some practical situations; for instance, there may be more than one failure occurring in a computation interval.

Due to the downside of the model in Yang [21], Daly [22] proposed a higher order failure model to estimate the optimum check-point interval. The model of Daly [22] assumes that there may be more than one failure occurring in a computation interval, which is closer to the realistic situation. The optimum computation interval figured out by Daly [22] is

$$\varphi_{opt} = \begin{cases} \sqrt{2\delta M}\left[1 + \frac{1}{3}\left(\frac{\delta}{2M}\right)^{1/2} + \frac{1}{9}\left(\frac{\delta}{2M}\right)\right] - \delta & \text{if } \delta < 2M, \\ M & \text{if } \delta \geq 2M. \end{cases} \quad (1)$$

$$\varphi_{opt} = \begin{cases} \sqrt{2\delta M} - \delta & \text{if } \delta < \frac{1}{2}M, \\ M & \text{if } \delta \geq \frac{1}{2}M. \end{cases} \quad (2)$$

which is a good rule of thumb for most practical systems.

However, the models in both Yang [22] and Daly [22] are aimed at one task running on a processor, which are not applicable to DAG based application running on cloud for the following reasons. First, there are many subtasks running on different processors; the completion time of each subtask may have influence on the successive subtasks. Second, the importance of each subtask in a DAG (a DAG based application) is different; for instance, the subtasks on the critical path of the DAG are more important than the others. Therefore, some scholars proposed heuristic methods aiming at DAG based application running on cloud.

Aiming to resolve the fault tolerance for DAG based application running on cloud, Zhu [23] and Qin [24] proposed copy task based methods. In general, the basic idea of copy task based methods is running an identical copy of each subtasks on different processors, the subtasks and their copies can be mutually excluded in time. However, these approaches assume that tasks are independent of one other, which cannot meet the needs of real-time systems where tasks have precedence constraints. In [24], for given two tasks, the authors defined the necessary conditions for their backup copies to safely overlap in time with each other and proposed a new overlapping scheme named eFRD (efficient fault-tolerant reliability-driven algorithm), which can tolerate processors failures in a heterogeneous system with fully connected network.

In Zhu [23], on the basis of Qin [24], the authors established a real-time application fault-tolerant model that extends the traditional copy task based model by incorporating the cloud characteristics. Based on this model, the authors developed approaches for subtask allocation and message transmission to ensure faults can be tolerated during the application execution and proposed a dynamic fault tolerant scheduling algorithm, named FASTER (fault tolerant scheduling algorithm for real-time scientific workflow). The experiment results show that the FASTER is better than eFRD [24].

Unfortunately, the disadvantage of copy task based methods including Zhu [23] and Qin [24] is very conspicuous. First, the copy of each subtask may consume more resources on the cloud, which makes them uneconomical. Second, the copy of each subtask will be executed only when the original subtask failed; this process will waste a lot of time, and it will be even worse due to the DAG based application; i.e., the deadline of the application will be not guaranteed in most of cases if the deadline is near to the critical path.

Thus, in this paper, we will combine the advantage of theoretic methods and heuristic methods to propose a novel fault tolerance algorithm for a big data application running on cloud.

Table 1: Cost optimization factors.

Symbols	Definitions
$T = G(V, E)$	a scientific application
$G(V, E)$	DAG of the tasks in T
CP	the critical path of $G(V, E)$
τ_i	the i-th subtask in V
ω_i	the weight of i-th tasks in V
V	the set of τ_i in T
$e_{i,j}$	data dependence from τ_i to τ_j
E	the set of $e_{i,j}$
δ	the time to make a check-point file
φ_{opt}	the optimal check-point interval
$T_W(\varphi)$	the practical completion time for the task
R	the time to read a check-point file
X	the time before a failure in an interval
T_c	the computation time for a task
S	a schedule which maps the tasks to processors
$wt(\tau_i)$	the weight of τ_i
$pred(\tau_i)$	the predecessor subtasks set of τ_i in $G(V, E)$
$succ(\tau_i)$	the successors subtasks set of τ_i in $G(V, E)$
$succP(\tau_i)$	the next subtask τ_i on the same processor
$ft(\tau_i)$	the finish time of τ_i on the S
$estS(\tau_i)$	the earliest start time of τ_i on the S
$lftS(\tau_i)$	the latest finish time of τ_i on the S
$slackT(\tau_i)$	the slack time of τ_i on the S
$\overline{slackT}(\tau_i)$	the indirect slack time of τ_i on the S
$\alpha(\tau_i)$	denotes the quantitative importance of τ_i
$\overline{\alpha}$	The threshold of α
$KeyT$	The set of key subtasks
$Ms(T)$	The makespan of the application T

3. Models and Formulation

In this section, we introduce the models of big data application, cloud platform, and the failure model then formally define the problem this paper is addressing. To improve the readability, we sum up the main notations used throughout this paper in Table 1.

3.1. Big Data Application Model. The model of a big data application T is denoted by $G(V, E)$, where $G(V, E)$ represents a DAG. Besides, we use T_c to denote the execution time of the critical path in $G(V, E)$. Each node $n_i \in V$ represents a subtask τ_i ($1 \leq i \leq v$) of T, and v is the total number of subtasks in T. W is the set of weights, in which each wire presents the execution time of a subtask n_i on a VM. E is the set of edges in $G(V, E)$, and an edge (n_i, n_j) represents the dependence between n_i and n_j; i.e., a task can only start after all its predecessors have been completed.

Figure 1 shows an example of DAG for a big data workflow, consisting of twelve tasks from τ_1 to τ_{12}. The DAG vertices related to tasks in T are represented by circles, while the directed edges denote the data dependencies among the tasks.

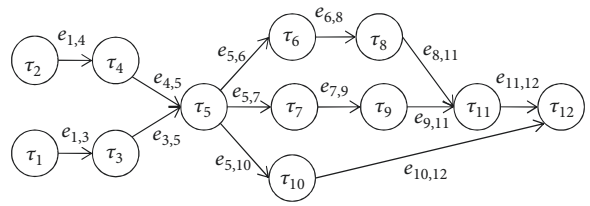


Figure 1: A DAG based application.

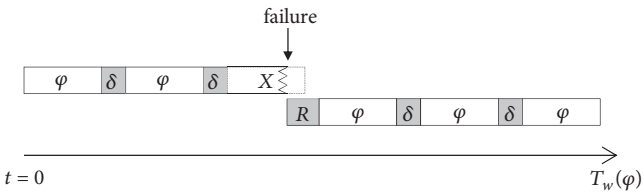


Figure 2: An example of a failure.

3.2. Cloud Platform Model. A cloud platform C is modeled as a set of processors $\{P_1, P_2, \ldots, P_{\mathbb{N}}\}$ and $\mathbb{N}$ is the total number of processors on the cloud. We use N to denote the number of processors rented by users for executing an application. Actually, the $\mathbb{N}$ is much greater than the mount which the user need. In general, to reduce the cost (monetary cost or time cost), the users apply proper schedule arithmetic to deploy their big data applications on cloud. But most of schedule algorithms did not consider the failure in each processor, which may consume extra cost (monetary cost and time cost).

3.3. The Check-Point Model. The check-point technology has been used on cloud for years, which makes application complete successfully in the shortest amount of time. In general, the check-point model is defined as bellow: ideally, the time to complete a task on a processor is denoted by T_c; we use φ to denote the check-point interval. After each computation interval (φ), the processor makes a backup for the current status of the system, and the time consumed by this process is represented by δ. If a failure occurs, the time consumed to read the latest backup and restart the computation is denoted by R. Finally, we use $T_W(\varphi)$ to denote the practical completion time for the task running on a processor while the check-point interval is φ.

Referring to Figure 2, the ideal completion time for the task is $T_c = 5\varphi$. Actually, there is a failure occurring after X time in the third interval, and it takes the processor R time to restart the third interval. At last, the practical time consumed by the tasks is $T_W(\varphi) = 5\varphi + R + X + 4\delta$.

3.4. The Failure Model. For a given MTBF (mean time between failures) which is denoted by M, according to [21], the life distribution model for mechanical and electrical equipment is described by an exponential model. Thus, the probability density function is

$$f(t) = \frac{1}{M} e^{-t/M}. \tag{3}$$

Then, the probability of a failures occurring before time Δt for a processor is represented by a cumulative distribution function

$$P(t \le \Delta t) = \int_{\Delta t}^{0} \frac{1}{M} e^{-t/M} dt = 1 - e^{-\Delta t/M}. \tag{4}$$

Obviously, the probability of successfully completing for a time Δt without a failure is

$$P(t > \Delta t) = 1 - P(t \le \Delta t) = e^{-\Delta t/M}. \tag{5}$$

We use T_c to denote the computation time for a subtask running on a processor; the φ denotes the compute interval between two check-points. Then, the average number of attempts (represented by $No.a$) needed to complete T_c is

$$No.a = \frac{T_c/\varphi}{P(t > \Delta t)} = \frac{T_c e^{\Delta t/M}}{\varphi}. \tag{6}$$

Therefore, the total number of failures during Δt is the number of attempts minus the number of successes.

$$n(\Delta t) = \frac{T_c e^{\Delta t/M}}{\varphi} - \frac{T_c}{\varphi} = \frac{T_c}{\varphi}\left(e^{\Delta t/M} - 1\right). \tag{7}$$

Notice that this assumes that we will never have more than a single failure in any given computation interval. Obviously, this assumption is relaxed in a real-life scenario. Thus, in [22], the scholar presented a multiple failures model.

$$n(\varphi) = \frac{(T_c - \delta + \delta T_c/\varphi)}{M - \{E(\varphi + \delta) + R\} P(\varphi) - E(R + \varphi + \delta)[1 - P(\varphi)]}, \tag{8}$$

where

$$E(\varphi + \delta) = M + \frac{\varphi + \delta}{1 - e^{(\varphi+\delta)/M}}$$
$$E(R + \varphi + \delta) = M + \frac{R + \varphi + \delta}{1 - e^{(\varphi+\delta)/M}} \tag{9}$$
$$P(\varphi) = e^{-(R+\varphi+\delta)/M}$$

The derivation process of Formula (8) is detailed in [22]; we will not repeat the process in this paper. In this paper, we will take Formula (8) as the failure model in our framework for two reasons; first, this could only provide the MTBF (M, mean time between failures) determined by statistics [19, 25] and, second, this model is closer to reality than the other model in [21].

3.5. Definitions. In order to make readers have a better understanding of this algorithm, we make some definitions first. For a given big data application $T = G(V, E)$ and a schedule S, we define the following terms.

3.5.1. Schedule (S). A schedule S is a map from the subtasks in $G(V, E)$ to the processors on the cloud; meanwhile, the start time and the finish time of each subtask have been figured out. In general, the S is determined by a static schedule algorithm, such as HEFT [26] and MSMD [27].

3.5.2. Weight ($wt(\tau_i)$). The $wt(\tau_i)$ is the weight (execution time) of τ_i running on a processor.

3.5.3. Predecessors ($pred(\tau_i)$). For a subtask τ_i, its predecessors subtask set is defined as below:

$$pred(\tau_i) = \left\{tau_j \mid \tau_j \in V \wedge (\tau_i, \tau_j) \in E\right\}. \quad (10)$$

3.5.4. Successors ($succ(\tau_i)$). For a subtask τ_i, its successors subtask set is defined as below:

$$pred(\tau_i) = \left\{tau_j \mid \tau_j \in V \wedge (\tau_j, \tau_i) \in E\right\}. \quad (11)$$

3.5.5. Successor on the Processor ($SuccP(\tau_i)$). For a given schedule S, the $SuccP(\tau_i)$ of τ_i is the set of the next subtask deployed on the same processor.

3.5.6. Finish Time ($ft(\tau_i)$). The $ft(\tau_i)$ denotes the finish time of τ_i on the schedule S.

3.5.7. Earliest Start Time on the Schedule ($estS(\tau_i)$). For a given schedule S, the start time of τ_i is the $estS(\tau_i)$. It should be noted that the $estS(\tau_i)$ is different with the traditional earliest start time in DAG.

3.5.8. Latest Finish Time on the Schedule ($lftS(\tau_i)$). For a given schedule S, the $lftS(\tau_i)$ of τ_i is defined below:

$$lftS(\tau_i) = \begin{cases} ft(\tau_i), & \text{if } succ(\tau_i) \text{ and } SuccP(\tau_i) \text{ are empty,} \\ \min\limits_{\tau_j \in succ(\tau_i) \vee SuccP(\tau_i)} \{estS(\tau_i)\}, & \text{otherwise.} \end{cases} \quad (12)$$

Same with $estS(\tau_i)$, the $lftS(\tau_i)$ is different with the traditional earliest start time in DAG.

3.5.9. Slack Time ($slackT(\tau_i)$). For a given schedule S, the $slackT(\tau_i)$ of τ_i is defined as follows:

$$slackT(\tau_i) = lftS(\tau_i) - estS(\tau_i) - weight(\tau_i). \quad (13)$$

3.5.10. Indirect Slack Time ($\overline{slackT}(\tau_i)$). For a given schedule S, the $\overline{slackT}(\tau_i)$ of τ_i is defined as follows:

$$\overline{slackT}(\tau_i) = \begin{cases} slackT(\tau_i), & \text{if } slackT(\tau_i) \neq 0, \\ \min\left\{\dfrac{wt(\tau_i)}{wf(\tau_i) + wt(\tau_j)} \overline{slackT}(\tau_j)\right\}, & \text{else if } succ(\tau_i) \vee SuccP(\tau_i) = \emptyset, \\ 0, & \text{otherwise.} \end{cases} \quad (14)$$

The $\overline{slackT}(\tau_i)$ denotes that the slack time of a subtask can be shared by its predecessors.

3.5.11. Importance ($\alpha(\tau_i)$). The $\alpha(\tau_i)$ denotes the quantitative importance of the subtask τ_i in the schedule S. The $\alpha(\tau_i)$ is calculated by

$$\alpha(\tau_i) = \frac{\overline{slackT}(\tau_i)}{wt(\tau_i)}. \quad (15)$$

3.5.12. Threshold of Importance ($\overline{\alpha}$). The threshold $\overline{\alpha}$ denotes whether a subtask is a key subtask; i.e., if $\alpha(\tau_i) < \overline{\alpha}$, then the subtask τ_i is a key subtask.

3.5.13. The Set of Key Subtask (KeyT). The signal *KeyT* denotes the set of key subtask for a given schedule S.

3.6. Problem Formalization. The ultimate objective of this work is to provide a high-performance fault tolerance mechanism and make sure that the proposed fault tolerance mechanism will consume less computation cost and makespan. The computation cost represents the processor time consumed by all the subtasks in T; thus, the object of minimizing computation cost is defined by

$$\text{Minimize} \quad \sum_{\tau_i \in T} T_W(\tau_i). \quad (16)$$

The makespan can be defined as the overall time to execute the whole workflow by considering the finish time of the last successfully completed task. For an application T, this object is denoted by

$$\text{Minimize} \quad Ms(T). \quad (17)$$

4. The Fault Tolerance Algorithm

In this section, we first discuss the basic idea of our algorithm for the fault tolerance of running big data application on

cloud. Then, on the basis of the idea, we will propose the fault tolerance algorithm using Selective Mirrored Tasks Method.

4.1. The Basic Idea. As show in Section 2, the theoretic methods which are devoted to find the optimal φ_{opt} are not applicable to the DAG based application, even if the φ_{opt} they have determined is very accurate for one task running on a processor. Besides, the heuristic methods based on the copy task will waste a lot of extra resource, and the completion time of the application may be delayed by much more time.

To find a better solution, we will integrate the advantages of theoretic methods and heuristic methods to propose a high-performance and economical algorithm for big data application. The check-point method with an optimal computation interval φ_{opt} is a dominant and economical method for one task running on a processor; thus, the check-point mechanism is the major means in our approach. Furthermore, owing to the parallelism and the dependencies in a DAG based application, the importance of each subtask is different; i.e., the subtasks on the critical path are more important than the others. The fault tolerance performance of these subtasks which adopt check-point method is insufficient, because the completion time of an application depends to a great extent on the completion time of these subtasks. Therefore, for the important subtasks, we will improve the fault tolerance performance of an application by introducing the task copy based method. In the task copy based methods [24], the original task and the copy do not start at the same time; to reduce the completion time, the original task and the copy will start at the same time.

In summary, the basic idea is as follows. First, identify the important subtasks in the DAG based application, which are named as key subtasks in the rest of this article. Then, apply the task copy based methods to the key subtasks; meanwhile, all the subtasks will employ the check-point technology to improve the fault tolerance performance, but the key subtasks and the normal subtasks will use different optimal computation interval φ_{opt}, the details of which will be described in the following sections.

It should be noted that our fault tolerance algorithm will not schedule the subtasks on the processors; we just provide a fault tolerance mechanism based on the existing static scheduler algorithm (such as HEFT and MSMD) to make sure that the application can be completed with the minimum of time.

4.2. Fault Tolerance Algorithm Using Selective Mirrored Tasks Method (FAUSIT). Based on the basic idea above, we propose the FAUSIT to improve the fault tolerance for executing a large-scale big data application; the pseudocode of the FAUSIT is listed in Algorithm 1.

As shown in Algorithm 1, the input of FAUSIT is a map from subtasks to processors which is determined by a static scheduler algorithm and the output is fault tolerance operation determined by FAUSIT. The function **DetermineKeyTasks()** in Line (1) is to find the key subtasks according to the schedule S and the application T. Then, the function **DeployKeyTasks()** in Line (2) deploys the mirrored subtasks and determine the proper φ_{opt} for the subtasks.

Input: a schedule S of an application T
Output: a fault tolerance operation F
(1) **DetermineKeyTasks**$(S, T) \longrightarrow KeyT$
(2) **DeployKeyTasks**$(KeyT) \longrightarrow F$
(3) **return** F

Algorithm 1: Fault Tolerance Algorithm using Selective Mirrored Tasks Method (FAUSIT).

In the following content in this subsection, we will expound the two functions in FAUSIT.

*4.2.1. Function of **DetermineKeyTasks()**.* The function of **DetermineKeyTasks()** is to determine the key subtasks in the DAG based application. In order to make readers have a better understanding of this function, we need to expound the key subtask and the indirect slack time clearly.

Definition 1. Key subtask: in a schedule S, the finish time of a subtask has influence on the start time of its successors; if the influence exceeds a threshold, we define the subtask is a key subtask.

The existence of the key subtasks is very meaningful to our FAUSIT algorithm. For a given schedule S, in the ideal case, each subtask as well as the application will be finished in accordance with the S. In practice, a subtask may fail when it has executed for a certain time; then, the processor will load the latest check-point files for continuation. At this point, the delay produced by the failure subtask may affect the successors. For the subtasks which have sufficient slack time, the start time of the successors is free from the failed subtask. On the contrary, if the failed subtask has little slack time, it will affect the start time of the successors undoubtedly. Given all that, we need to deal with the key subtasks which has little slack time.

Definition 2. Indirect slack time: for two subtasks τ_i and τ_j, τ_j is the successor of τ_i, if τ_j has slack time (defined in Section 3.5.9), the slack time can be shared by τ_i, and the shared slack time is indirect slack time for τ_i.

The indirect slack time is a useful parameter in our FAUSIT algorithm, the existence of which will make the FAUSIT save a lot of time (makespan of the application) and cost. For a given schedule S, a subtask may have sufficient slack time which can be shared by predecessors. Thus, the predecessors may have enough slack time to deal with failures; then, the completion time of the predecessors and the subtask will not delay the makespan of the application. Indeed, the indirect slack time is the key parameter to determine whether a subtask is a key subtask. Moreover, the indirect slack time reduces the count of the key subtasks in a big data application, which will save a lot of cost, because the key subtask will apply mirrored task method.

Input: a schedule S of an application T, $\overline{\alpha}$.
Output: the key subtask set $KeyT$.
(1) Determine the $ft()$, $estS()$ and the $lftS()$ of the subtasks in T according to S.
(2) Determine the $slackT()$ of each subtask.
(3) Determine the $\overline{slackT}()$ of the subtasks by recursion.
(4) Determine the set of key subtasks according to $\alpha \longrightarrow KeyT$
(5) **return** $KeyT$.

Algorithm 2: DetermineKeyTasks().

The pseudocode for function **DetermineKeyTasks()** is shown in Algorithm 2. The input of **DetermineKeyTasks()** is a schedule S of an application T and the threshold $\overline{\alpha}$; the output is the set of key subtasks. First, in Line (1), the $ft()$, $estS()$ and the $lftS()$ of the subtasks are determined according to Sections 3.5.6 and 3.5.7 and Formula (12). Then, the $slackT()$ of each subtask is figured out by Formula (13) in Line (2). Line (3) determines the $\overline{slackT}()$ of the subtasks by recursion. Finally, the set of key subtasks is determined according to the threshold α.

Table 2 shows the process of **DetermineKeyTasks()**. When the $\overline{\alpha} = 0.15$, Figure 3 shows the key subtasks which shall adopt the mirrored task method to improve the performance of fault tolerance.

It should be noted that the threshold $\overline{\alpha}$ is given by the users, which is related to the makespan of the application; i.e., the higher $\overline{\alpha}$ leads to more key subtasks. Then, the makespan of the application is shorter. On the contrary, the smaller $\overline{\alpha}$ will lead to a longer makespan.

*4.2.2. Function of **DeployKeyTasks()**.* The function of **DeployKeyTasks()** is to deal with the key subtasks, which minimizes the makespan of the application to the least extent. The main operation of **DeployKeyTasks()** is using mirrored task method; in order to make readers have a better understanding of this function, we need to expound the mirrored task method first.

The mirrored task method is to deploy a copy of a key subtask on another processor; the original subtask is denoted by τ_i^P and the copy of the subtask is denoted by τ_i^C. The τ_i^P and the τ_i^C start at the same time, and the check-point interval of them is $2\varphi_{opt}$ (the φ_{opt} is determined by [22]). The distinction between the τ_i^P and the τ_i^C is that the first check-point interval of τ_i^P is φ_{opt}; meanwhile, the first check-point interval of τ_i^C is $2\varphi_{opt}$. Obviously, once a failure occurs in one of the processors, the interlaced check-point interval of the two same subtasks makes sure that the time delayed by dealing with the failure is W (the time to read a check-point file).

Figure 4 shows an example of the mirrored task method. The Figure 4(a) displays the ideal situation of the mirrored task method; i.e., there are no failures happen in both P_k and P_l, and the finish time is $4\varphi + 2\delta$. In Figure 4(b), there is only one failure happening on P_l in the second check-point interval φ^2. First, the processor P_l reads the latest check-point file named δ_k^1. Then, with time goes by, the processor P_l will immediately load the latest check-point file δ_k^3 when it generates. Thus, the finish time is $4\phi + 2\delta + W$. Figure 4(c) illustrates the worst case; both P_k and P_l have a failure in the same interval φ^2, the two processors will have to load the latest check-point file δ_k^1. Thus, the finish time is $4\varphi + 2\delta + W + X$.

Obviously, the mirrored task method is far better than the traditional copy task based method, since the copy task will start only when the original task failed, and it will waste a lot of time. Moreover, the mirrored task method is also better than the method in [22], since the probability of the worst case is far less than the probability of the occurrence for one failure in a processor.

The pseudocode for function **DeployKeyTasks()** is displayed in Algorithm 3. The loop in Line (1) makes sure that all the key subtasks can be deploy on the processors. The applied processors should be used first when deploying the key subtasks; the loop in Line (2) illustrates this constraint.

Line (3) makes sure that the key subtasks τ_i^B have no overlapping with other tasks on P_j. Lines (4)-(5) deploy τ_i^B on P_j. If all the applied processors have no idle interval for τ_i^B (Line (8)), we will have to apply a new processor and deploy τ_i^B on it; then, we put the new processor into the set of applied processors (Lines (9)-(11)). At last, we deploy the check-point interval (φ_{opt} or $2\varphi_{opt}$) to the processors (Line (14)) and save these operations in T (Line (15)).

It should be noted that the overlapping in Line (3) is not just the computation time of the τ_i^B; we also consider the delayed time which may be caused by failures. In order to avoid the overlap caused by the delayed τ_i^B, we use the $1.3w_i$ as the execution time of τ_i^B to determine whether an overlap happen, since the $1.3w_i$ is much greater than a delayed τ_i^B.

4.3. The Feasibility Study of FAUSIT. The operations to the processors in our FAUSIT are complex, such as the different check-point interval and the mirrored subtasks; the readers may doubt the feasibility of FAUSIT. Thus, we will explain the feasibility study of FAUSIT in this subsection.

According to [15], Google Cloud provides the *gcloud* suit for users to operate the processors (virtual machine) remotely. The operations of *gcloud* include (but not limited to) applying processors, releasing processors, check-point,

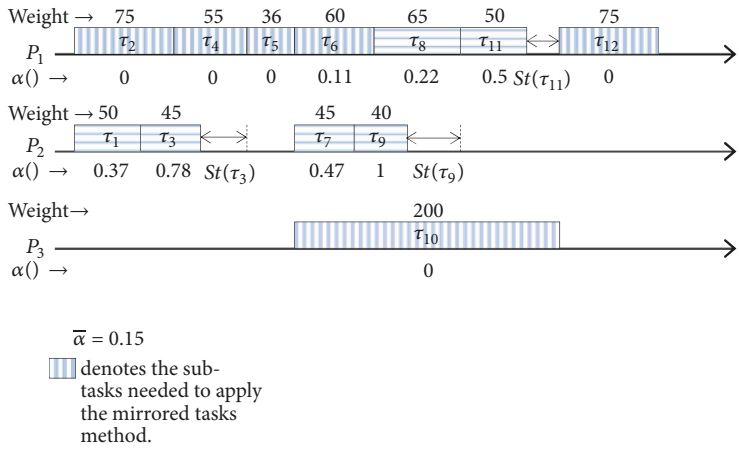
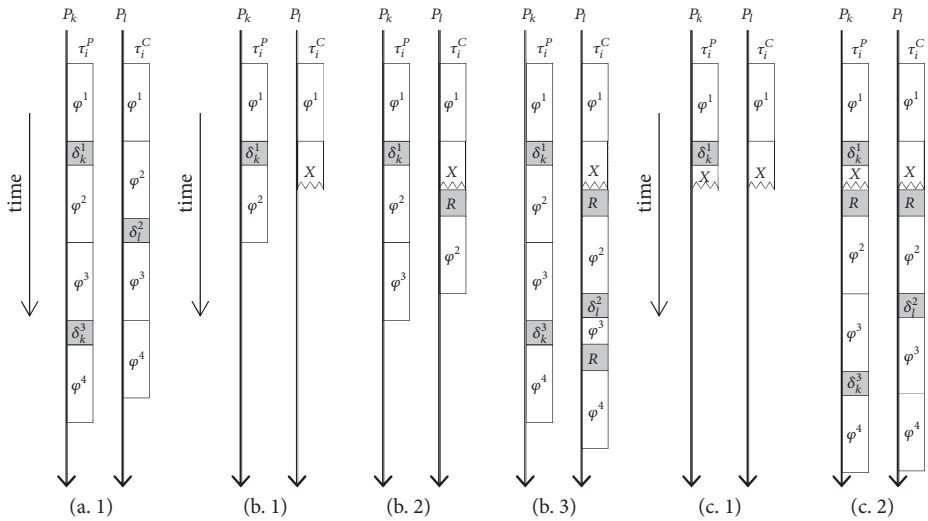


Figure 3: An example of $\overline{\alpha} = 0.15$.



(a) The ideal situation (b) One failure occurs (c) Two failures occur simultaneously

Figure 4: An example of mirrored tasks method.

Table 2: An example of **DetermineKeyTasks()**.

Subtasks	Weight	$lft()$	$est()$	$St()$	$St'()$	$\alpha()$
τ_1	50	0	50	0	18.4	0.37
τ_2	75	0	75	0	0	0
τ_3	45	50	130	35	35	0.78
τ_4	55	75	130	0	0	0
τ_5	36	130	166	0	0	0
τ_6	60	166	226	0	6.8	0.11
τ_7	45	166	211	0	21.2	0.47
τ_8	65	226	291	0	14.1	0.22
τ_9	40	211	291	40	40	1
τ_{10}	200	166	366	0	0	0
τ_{11}	50	291	366	25	25	0.5
τ_{12}	75	366	441	0	0	0

Input: the key subtask set yT.
Output: a fault tolerance operation F
(1) **for** τ_i^B in $KeyT$ **do**
(2) **for** P_j in the processors which have been applied **do**
(3) **if** τ_i^B has no overlapping with other tasks on P_j **then**
(4) Deploy the τ_i^B on P_j
(5) End **for** in Line (2).
(6) **end if**
(7) **end for**
(8) **if** τ_i^B has not be deployed on the processor. **then**
(9) Apply a new processor.
(10) Deploy τ_i^B on the new processor.
(11) Put the new processor in the set of processors.
(12) **end if**
(13) **end for**
(14) Deploy the check-point interval to the processors.
(15) Save the operations in F.
(16) **return** F

Algorithm 3: **DeployKeyTasks()**.

Table 3: The Characteristics of the Big Data Application.

Workflows	Task Count Range	Edge Count	Avg. Run Time of Task (Sec)	Deadlines Range
Epigenomics	100	322	3856.51	1.0CP
	200	644		
	...	...		
	1000	3228		

and loading check-point files. These operations in *gcloud* can make user implement the FAUSIT easily.

5. Empirical Evaluations

The purpose of the experiments is to evaluate the performance of the developed FAUSIT algorithm. We evaluate the fault tolerant of FAUSIT by comparing it with two other algorithms published in the literature. They are FASTER algorithm [23] and the method in [22]. The main differences of these algorithms to FAUSIT and their uses are briefly described below.

(i) FASTER: a novel copy task based fault tolerant scheduling algorithm. On the basis of copy task based method, the FASTER adjust the resources dynamically.

(ii) The method in [22]: it is a theoretic method which provides an optimal check-point interval φ_{opt}.

5.1. Experiment Settings. The DAG based big data applications we used for the evaluation are obtained from the DAG based applications benchmark provided by Pegasus WorkflowGenerator [28]. We use the largest application from the benchmark, i.e., Epigenomics, since the bigger application is more sensitive to failures. The detailed characteristics of the benchmark applications can be found in [29, 30]. In our experiments, the number of subtasks in an application is ranging from 100 to 1000. Since the benchmark does not assign deadlines for each application, we need to specify the deadlines; we assign a deadline for the applications: it is 1.0CP. Table 3 gives the characteristics of these applications including the count of tasks and edges, average task execution time and the deadlines.

Because FAUSIT does not schedule the subtasks on the processors, we hire a high-performance schedule algorithm (MSMD) to determine the map from subtasks to the processors. MSMD is a novel static schedule algorithm to reduce the cost and the makespan for an application. On the basis of MSMD, we use FAUSIT to improve the fault tolerant of an application.

5.2. Evaluation Criteria. The main objectives of the FAUSIT are to find the optimal fault tolerant mechanism for a big data application running on cloud. The first criterion to evaluate the performance of a fault tolerant mechanism is how much time is delayed to complete the application, i.e., the makespan to finish the application. We introduce the concept of *makespan delay rate* to indicate how much extra time consumed by the fault tolerant mechanism. It is defined as follows:

$$Mak.Del.Rate = \frac{\text{The practical makespan}}{\text{The ideal makespan}}. \quad (18)$$

where the ideal makespan represents the completion time for running an application without fault tolerant mechanism and any failures, and the practical makespan is the completion time consumed on practical system which has a fault tolerant mechanism and the probability of failures.

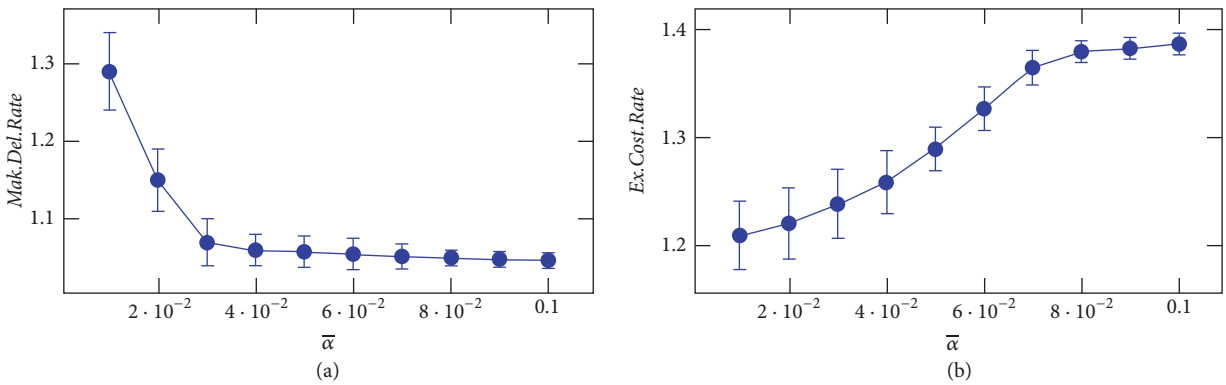


Figure 5: (a) The *Mak.Del.Rate* and (b) the *Ex.Cost.Rate* of $\overline{\alpha}$.

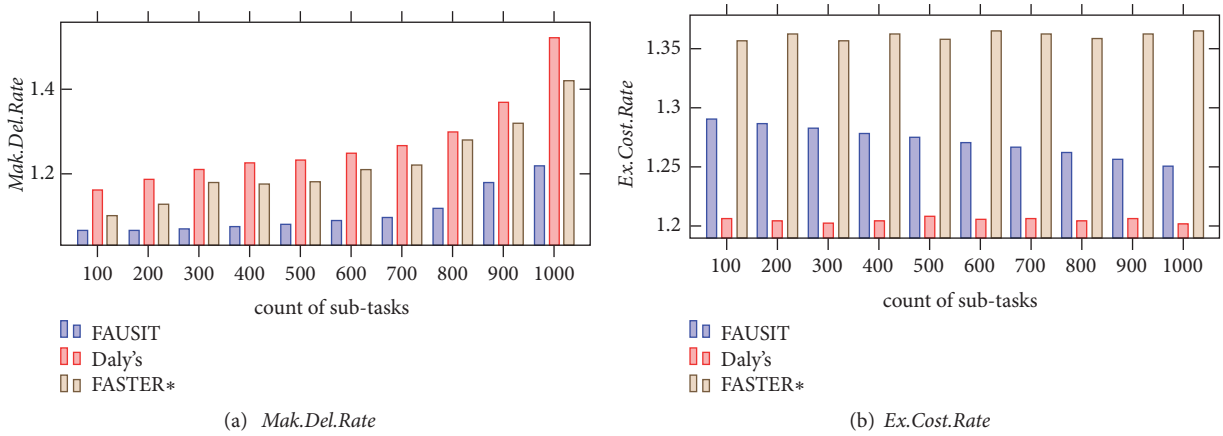


Figure 6: The result of *Mak.Del.Rate* and *Ex.Cost.Rate*.

The second goal of the FARSIT is to minimize the extra cost consumed by the fault tolerant mechanism. Undoubtedly, to improve the performance of fault tolerant, any fault tolerant mechanisms will have to consume extra computation time for an application. Thus, the extra computation time is a key criterion to evaluate the performance of fault tolerant mechanism. Therefore, we define extra cost rate for the evaluation:

$$Ex.Cost.Rate = \frac{\text{The practical processors time}}{\text{The ideal processors time}}. \quad (19)$$

where the practical processors time denotes the time consumed by the processors to running an application for a practical system with failures, The ideal processors time is the sum of the w_i in $G(V, E)$ for an application.

5.3. The Optimal α for FAUSIT. Before comparing with the other algorithms, we need to determine the optimal $\overline{\alpha}$ for FAUSIT first. Due to the optimal $\overline{\alpha}$ is an experimental parameter, we have to figure it out by experiments.

We test the $\overline{\alpha}$ by the Epigenomics application with 1000 subtasks for 10 times and make the $T_d = 1.0T_c$; the results are shown in Figures 5(a) and 5(b). In Figure 5(a), the *Mak.Del.Rate* becomes lower alone with the increased; i.e., a larger $\overline{\alpha}$ will lead to a shorter makespan for an application. On the contrary, Figure 5(b) shows that the larger $\overline{\alpha}$ will lead to more cost for an application.

Through a comprehensive analysis of Figures 5(a) and 5(b), we make the $\overline{\alpha} = 0.033$ which can equalize the *Mak.Del.Rate* and the *Ex.Cost.Rate* and make sure that both the *Mak.Del.Rate* and the *Ex.Cost.Rate* are smaller than other situations.

5.4. Comparison with the Other Solutions. Since the proposed FAUSIT algorithm is a heuristic algorithm, the most straightforward way to evaluate its performance is to compare with the optimal solution when possible. We randomly generate 10 different applications for each scale of Epigenomics shown in Table 3, and we make the $T_d = 1.0CP$.

What should be paid attention to is that although the FASTER is a copy tasks based method, but it is developed for multiple DAG based big data applications. As a result, it cannot compare the performance of it with the other two methods directly. In order to make a comprehensive comparison, on the basis of FASTER, we modified the FASTER to make it handle a single DAG based application, and this new FASTER is denoted by FASTER∗.

The average of *Mak.Del.Rate* and *Ex.Cost.Rate* are displayed in Figure 6. In Figure 6(a), our FAUSIT has the minimum *Mak.Del.Rate* for any size of application. Meanwhile,

the FASTER∗ has higher *Mak.Del.Rate* than our FAUSIT, and the Daly's method has the highest *Mak.Del.Rate*. The result in Figure 6(a) shows that our FAUSIT has the best performance for minimizing the makespan of an application among these methods.

In Figure 6(b), due to the fact that the Daly's method only adopts the check-point mechanism, this makes it has the minimum *Ex.Cost.Rate*. Meanwhile, the FASTER∗ consumes the maximum cost, since the *Ex.Cost.Rate* of it is very large. Interestingly, along with the increase of the count of subtasks in an application, the *Ex.Cost.Rate* of our FAUSIT is becoming smaller, which indicates that our FAUSIT has the ability to handle much bigger scale of application.

In conclusion, compared with the FASTER, our FAUSIT outperforms FASTER on both *Mak.Del.Rate* and *Ex.Cost.Rate*. Compared with the Daly's method, our FAUSIT outperforms it much more on *Mak.Del.Rate* and only consume 9% extra cost, which still makes our FAUSIT have competitiveness in practical system. Besides, the $\overline{\alpha}$ in our FAUSIT can satisfy the requirements from different users; i.e., if the users need a shorter makespan for an application, they can turn the $\overline{\alpha}$ up. On the contrary, if the users care about the cost, they can turn the $\overline{\alpha}$ down. Thus, the $\overline{\alpha}$ make the FAUSIT have strong usability.

6. Conclusion

This paper investigates the problem of improving the fault tolerant for a big data application running on cloud. We first analyze the characteristics of running a task on a processor. Then, we present a new approach called selective mirrored task method to deal with the imbalance between the parallelism and the topology of the DAG based application running on multiple processors. Based on the selective mirrored task method, we proposed an algorithm named FAUSIT to improve the fault tolerant for a big data application running on cloud; meanwhile, the makespan and the computation cost is minimized. To evaluate the effectiveness of FAUSIT, we conduct extensive simulation experiments in the context of randomly generated workflows which are real-world application traces. Experimental results show the superiorities of FAUSIT compared with other related algorithms, such as FASTER and Daly's method. In our future work, due to the superiorities of selective mirrored task method, we will try to apply it to other big data applications processing scenarios, such as improving the fault tolerant of multiple applications in the respect of could providers. Furthermore, we will also investigate the effectiveness of the selective mirrored task method in parallel real-time applications running on cloud.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported partly by Doctoral Research Foundation of Liaoning under grant no. 20170520306, Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis Open Fund (HCIC201605), Guangxi Youth Capacity Improvement Project (2018KY0166), and Supercomputing Center of Dalian University of Technology.

References

[1] M. Armbrust, A. Fox, R. Griffith et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] Market Research Media, "Global cloud computing market forecast 2019-2024," https://www.marketresearchmedia.com/?p=839.

[3] L. F. Sikos, "Big data applications," in *Mastering Structured Data on the Semantic Web*, 2015.

[4] G.-H. Kim, S. Trimi, and J.-H. Chung, "Big-data applications in the government sector," *Communications of the ACM*, vol. 57, no. 3, pp. 78–85, 2014.

[5] G. Wang, T. S. E. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*, pp. 103–108, ACM, Helsinki, Finland, August 2012.

[6] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012.

[7] H. Liu, H. Ning, Y. Zhang, Q. Xiong, and L. T. Yang, "Role-dependent privacy preservation for secure V2G networks in the smart grid," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 2, pp. 208–220, 2014.

[8] E. Sindrilaru, A. Costan, and V. Cristea, "Fault tolerance and recovery in grid workflow management systems," in *Proceedings of the 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '10)*, pp. 475–480, IEEE, February 2010.

[9] Q. Zheng, "Improving MapReduce fault tolerance in the cloud," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, IEEE, April 2010.

[10] L. Peng et al., "Deep convolutional computation model for feature learning on big data in internet of things," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 2, pp. 790–798, 2018.

[11] Q. Zhang, L. T. Yang, Z. Yan, Z. Chen, and P. Li, "An efficient deep learning model to predict cloud workload for industry informatics," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3170–3178, 2018.

[12] L. Man and L. T. Yang, "Hybrid genetic algorithms for scheduling partially ordered tasks in a multi-processor environment," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, IEEE, 1999.

[13] Q. Zhang, L. T. Yang, and Z. Chen, "Deep computation model for unsupervised feature learning on big data," *IEEE Transactions on Services Computing*, vol. 9, no. 1, pp. 161–171, 2016.

[14] "Cluster networking in ec2," https://amazonaws-china.com/ec2/instance-types.

[15] "Google cloud," https://cloud.google.com/.

[16] "Microsoft azure," https://azure.microsoft.com/.

[17] J. Rao, Y. Wei, J. Gong, and C.-Z. Xu, "QoS guarantees and service differentiation for dynamic cloud applications," *IEEE Transactions on Network and Service Management*, vol. 10, no. 1, pp. 43–55, 2013.

[18] J. Wentao, Z. Chunyuan, and F. Jian, "Device view redundancy: An adaptive low-overhead fault tolerance mechanism for many-core system," in *Proceedings of the IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC '13)*, IEEE, 2013.

[19] M. Engelhardt and L. J. Bain, "On the mean time between failures for repairable systems," *IEEE Transactions on Reliability*, vol. 35, no. 4, pp. 419–422, 1986.

[20] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: towards scalable large instruction window processors," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 423–434, IEEE Computer Society, San Diego, Calif, USA, 2003.

[21] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530-531, 1974.

[22] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[23] X. Zhu, J. Wang, H. Guo, D. Zhu, L. T. Yang, and L. Liu, "Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3501–3517, 2016.

[24] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems," *Parallel Computing. Systems & Applications*, vol. 32, no. 5-6, pp. 331–356, 2006.

[25] S. Mu, M. Su, P. Gao, Y. Wu, K. Li, and A. Y. Zomaya, "Cloud storage over multiple data centers," *Handbook on Data Centers*, pp. 691–725, 2015.

[26] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[27] H. Wu, X. Hua, Z. Li, and S. Ren, "Resource and instance hour minimization for deadline constrained DAG applications using computer clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 885–899, 2016.

[28] G. Juve, "workflowgenerator," https://confluence.pegasus.isi.edu/display/pegasus/workflowgenerator, 2014.

[29] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi, "Characterization of scientific workflows," in *Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS '08)*, pp. 1–10, IEEE, November 2008.

[30] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.