Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 2170 – 2185

doi:10.3906/elk-1806-232

Task graph scheduling in the presence of performance fluctuations of computational resources

Najmeh MALAKOUTIFAR[1], **Hassan MOTALLEBI**[2,*]

[1]Department of Computer Engineering, Islamic Azad University, Kerman Branch, Kerman, Iran

[2]Department of Electrical and Computer Engineering, Graduate University of Advanced Technology, Kerman, Iran

Received: 30.06.2018 • **Accepted/Published Online:** 22.12.2018 • **Final Version:** 15.05.2019

Abstract: Most of the existing work in the area of task graph scheduling considers resources with fixed processing capacity. The algorithms in these works rely on an estimation of the execution times of tasks on different resources. However, in practice, due to fluctuations in performance of cloud resources, these algorithms have challenges in these environments. In this paper, we focus on the problem of fault-tolerant scheduling of task graphs in the presence of performance fluctuations of computational resources. With the aim of reducing the adverse impacts of both soft errors and resource performance degradations, we propose an opportunistic task replication scheme that uses idle durations of resources for replicating tasks. Unlike the previous works, the proposed algorithm does not rely on estimation of task execution times for finding idle resources. We introduce the notion of concurrency graphs and propose a graph theory-based algorithm for finding the number of idle resources during the execution of a set of tasks. The appropriate redundancy for each task is chosen with respect to the number of idle resources and the characteristics of the set of tasks that are being processed concurrently. Simulation experiments show that, in most situations, the proposed algorithm outperforms the previous algorithms in terms of average execution time and cost.

Key words: Task graph scheduling, resource performance fluctuations, concurrency graph, Markov-modulated Poisson process, performance-varying resource.

1. Introduction

Precedence-constrained parallel applications, also known as task graphs (workflows), constitute a common model for describing a wide range of scientific applications in distributed systems. A task graph is an abstract representation of an application in the form of a directed acyclic graph (DAG), whose vertices represent tasks and edges represent the possible precedence constraints and data dependencies among them [1]. Task graph scheduling is the problem of mapping tasks to appropriate resources and of ordering tasks on each resource such that the user's requirements are met. The characteristics of cloud infrastructures make them a suitable platform for execution of task graphs [2]. In this paper, we investigate the problem of fault-tolerant scheduling of task graphs in a cloud environment where computational resources are both failure-prone and performance-varying.

A large body of work has examined the problem of task graph scheduling in distributed environments such as clouds [1–4]. The algorithms in almost all of these works rely on estimation of execution times of tasks on different resources, which are obtained through analysis of historical data. These algorithms make overly optimistic assumptions about the performance exhibited by the underlying cloud infrastructure [2]. Due to

*Correspondence: h.motallebi@kgut.ac.ir

technological and strategic factors, cloud environments do not have fixed execution time and performance. As reported in [5], fluctuations in performance of cloud resources can cause a variation of up to 30% for execution times and 65% for data transfer times. Fluctuations in performance of cloud resources delay task execution and delay in the execution of a task may lead to delay in the execution of its successors.

Besides the uncertainty rooted in resource performance fluctuations, we deal with another more important form of uncertainty, which is caused by possible failures in execution of tasks. In order to deal with such failures, two fundamental techniques are widely used: resubmission and replication [1, 6]. Resubmission (time redundancy) helps to mitigate failures by retrying the same task either on the same or another resource. In the replication technique (space redundancy), on the other hand, additional resources are provided to execute the same task to provide resilience. Since each of these techniques have their own benefits and drawbacks, some works [1] use a hybrid of the two techniques with the aim of achieving the benefits of both.

1.1. Motivation and contribution

In this paper, we focus on the problem of scheduling of task graphs in situations where computational resource are both failure-prone and performance-varying. Delay in execution of tasks, caused by either of these uncertainties, leads to tardiness in termination of the application. One possible way of dealing with this issue is task replication. For example, in order to mitigate the effects of performance degradation of resources and increase the probability of meeting deadlines, the algorithm in [2] tries to use idle durations of resources for replicating tasks. However, the algorithm has some drawbacks. First, idle durations of the resources are computed based on inaccurate estimates of task execution times, which are prone to significant variations. Second, the algorithm assumes a failure-free environment where performance degradation is the only cause of uncertainty in task execution times. However, in the presence of task failures, the uncertainty in task execution times is much larger and it is quite impractical to find idle durations of resources.

In this paper, we investigate the problem of task graph scheduling in a failure-prone environment in the presence of performance fluctuations of computational resources. In order to mitigate the impact of task failures and execution delays, we propose an opportunistic task replication scheme that uses idle durations of already provisioned resources to replicate tasks. In order to find resources with idle durations we propose a graph theory-based algorithm that, unlike the approach in [2], does not rely on estimation of task execution times. The proposed algorithm tries to minimize the fault-tolerance overhead by finding a compromising balance between the resubmission and replication techniques. The redundancies of tasks are chosen with respect to the number of idle resources and the characteristics of the set of tasks that are being processed.

One major problem with some existing (hybrid) redundancy selection algorithms [6–9] is the issue of excessive usage of replication redundancy, which causes the disadvantages of the resubmission technique, i.e. existing algorithms may use the computational power of a resource for processing backup copies of tasks while it would be better used for executing the primary copy of a task with higher criticality. While determining the appropriate redundancies of tasks, if the number of replicas of other concurrently executing tasks is not taken into consideration, we may encounter situations where task replicas are being processed serially rather than in parallel. This problem is especially more serious in situations where few resources are provisioned with the aim of reducing the cost.

The main idea of the CGOR algorithm is as follows: first, for each task, the set of possibly concurrent tasks is computed. Then, while a set of tasks is being processed, we determine the set of idle resources and try to apportion them among the set of active tasks according to their criticality, assigning more resources to more

critical tasks. The proposed concurrency graph-based opportunistic replication (CGOR) algorithm brings the following novelties:

- We propose a graph theory-based algorithm that finds, for each task, the set of possibly concurrent tasks, according to which we can determine the number of idle resources for executing additional replicas of tasks. For this, we introduce the notion of concurrency graphs. Unlike the method proposed in [2], in the proposed method, idle resources are identified according to the structure of the task graph and not based on our estimations of task execution times. Therefore, no matter how significant the uncertainties in task execution times are, the number of idle resources can be determined.
- We propose a new redundancy selection algorithm, which determines a near-optimal combination of the two redundancy techniques according to the number of resources with idle durations and the criticalities of active tasks. In order to avoid excessive replication of tasks, the proposed algorithm is designed to use idle resources for task replication.
- We use a Markov-modulated Poisson process-based performance fluctuation model for modeling the changes in performance of computational resources. We also propose a resource selection criterion for selecting resources with respect to their instantaneous processing capacities.

The performance of the CGOR approach is assessed using a simulation study and is compared against the prior state-of-the-art algorithms. We develop a random task graphs generation procedure for generating a synthetic workload. The simulation results, on randomly generated task graphs, show that the proposed approach has a promising performance with a significant improvement over the prior algorithms.

1.2. Structure of the paper

In Section 2, we present our model of the application and computational environment and the resource fluctuation model. In Section 3, we review some related work. Section 4 presents the proposed CGOR algorithm for scheduling task graphs for failure-prone and performance-varying resources. In Section 5, a comparison of the CGOR algorithm with some existing algorithms is given. Finally, Section 6 provides concluding remarks and future work.

2. Application and resource model

In this section, first we describe our model of the application and then we explain the computational environment and the resource performance fluctuation model.

2.1. Task graph model

A task graph is defined as follows:

Definition 1 ***(Task graph)***. *A precedence-constrained application is modeled as a task graph (DAG)* $\mathcal{W} = (\mathcal{T}, \mathcal{D})$, *where* $\mathcal{T} = \{t_1, \ldots, t_n\}$ *is a set of tasks and* $\mathcal{D} \subset \mathcal{T} \times \mathcal{T}$ *is the set of dependencies among them.*

An edge $d = (t_i, t_j)$ represents a precedence constraint which indicates that task t_i should finish executing before task t_j can start. To such an edge is assigned the size of the data that is transferred between t_i and

t_j. Also, to each node is assigned the size of the corresponding task. The set of predecessors ($pred(t_i)$) and successors ($succ(t_i)$) of a task t_i is defined as follows:

$$pred(t_i) = \{t_j \in \mathcal{T} \mid (t_j, t_i) \in \mathcal{D}\}, succ(t_i) = \{t_j \in \mathcal{T} \mid (t_i, t_j) \in \mathcal{D}\}. \quad (1)$$

A task t_i with $pred(t_i) = \emptyset$ ($succ(t_i) = \emptyset$) is called an entry (exit) task. In this paper, we always add dummy entry and exist tasks with zero execution time to the beginning and the end of the graph, respectively, to ensure that the given DAG has a single entry and a single exit task. These dummy tasks are connected with zero-weight arcs to the real entry and exit tasks.

2.2. Environment model

The computational environment is assumed to be a cloud service provider that offers different virtualized resources to its clients. In particular, we assume that the service provider offers several computational services $S = \{s_1, \ldots, s_m\}$ with different QoS parameters such as the type of the processor and memory size, and different prices as in [4]. The characteristics of the environment are specified using four parameters: 1) m, the number of resources; 2) P_{avg}, the average processing capacity; 3) C_{avg}, the average cost per billing period (30 min); and 4) λ_{avg}, the average failure rate. The processing capacity of each resource is uniformly chosen from $[0.5P_{avg}, 2P_{avg}]$ MIPS, i.e. the fastest resource is about four times faster than the slowest one and is approximately four times more expensive. We also have $C_{avg} = 0.5\$$. As in some previous works (e.g., [10]), the time to failure of resources is assumed to follow an exponential distribution and the rate of this distribution is chosen from a normal distribution with mean $\lambda_{avg} = 0.75 \times 10^{-3}$ and standard deviation $\sigma = 0.05 \times 10^{-3}$.

2.3. Performance fluctuation model

In our performance fluctuation model, the processing capacity of computational resources through time is adjusted according the rate assigned to the current state of a Markov-modulated Poisson process model. A Markov-modulated Poisson process (MMPP) is a doubly stochastic Poisson process whose rate is controlled by an irreducible continuous-time Markov chain (CTMC) [11]. A MMPP, which can be considered as a Poisson process with varying rate, is parameterized by a k-state CTMC with infinitesimal generator Q and k Poisson arrival rates $\Lambda = diag(C_0, \ldots, C_{k-1})$. We assume a performance fluctuation model with k performance modes (states) whose corresponding CTMC is shown in Figure 1. In this model, C_i, the processing rate of the resource in state $c_i, i = 0, \ldots, k-1$ is computed as follows:

$$C_i = C_0 - i \times \theta, i = 0, \ldots, k-1. \quad (2)$$

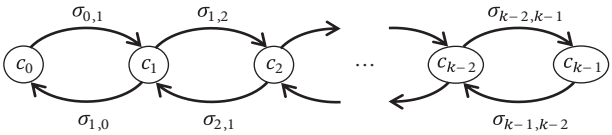


Figure 1. Markov chain corresponding to the MMPP-based performance fluctuation model.

As justified by the results in [3], we assume a performance fluctuation model with relatively continuous changes in processing capacity. Besides the continuous changes in the performance of resources, we also add some sudden changes in the performance. We assume that the performance level of the resource after a sudden

change is uniformly chosen from $\{c_0, \ldots, c_{k-1}\}$, and after an exponentially distributed period of time, X_f, the performance level is returned back to the same level as before. X_f is assumed to follow an exponential distribution with rate parameter δ_f. Also, the rate of the sudden changes is denoted by δ_s.

3. Related work

This section briefly summarizes some existing work on scheduling task graphs in distributed environments. Most of the research in this area is done with the assumption of a failure-free environment. However, some works address the problem in failure-prone environments [7, 8, 12, 13]. In [7], the authors proposed a fault-tolerant extension of the classic HEFT algorithm [14], which is based on replication redundancy. The works in [15, 16] proposed scheduling algorithms for service-oriented environments. In [4], the authors proposed an algorithm for scheduling deadline-constrained applications for an IaaS cloud environment. In order to meet the deadline, the overall deadline is distributed on the task graph nodes. The work in [2] also addressed the problem of scheduling of deadline-constrained applications. The authors considered the possibility of delays in task executions due to performance degradations. In order to reduce the impact of resource performance fluctuations on the deadline, the authors proposed to use the replication technique. In [17] the processing times of tasks were assumed to follow independent random variables. The authors proposed an energy-aware stochastic algorithm with the objective of optimizing execution time and energy consumption.

In order to provide fault-tolerance for long running tasks, some works used checkpointing redundancy. In [10, 18], the problem of determining the optimal checkpointing period was examined. In [19], in order to obtain the optimal solution for linear task graphs, a two-level checkpointing scheme was proposed, which combines the disk checkpointing technique with in-memory checkpoints. Since each of the redundancies have their benefits and drawbacks, some works [1, 6, 9, 20] used a hybrid of these complementary redundancies. The resubmission impact heuristic [6, 9] uses a combination of replication and resubmission. The number of replications of each task in this heuristic is determined based on the so-called resubmission impact metric. Finally, in [20], a hybrid of resubmission and checkpointing was proposed.

MMPP [11] models are widely used for traffic description and prediction and performance analysis of web and cloud [3, 21, 22]. In order to find model parameters, in [23] an EM algorithm was proposed, or computing the MLE estimates of the parameters of an m-state MMPP. The work in [24] investigated the issue of resource provisioning based on MMPP arrivals.

4. Concurrency graph-based opportunistic replication

In this section, we explain the proposed CGOR algorithm for selecting the appropriate redundancies for tasks. The main idea of the CGOR algorithm is that while a set $\mathcal{T}_{active}$ of tasks is being processed, we find the number of resources that remain idle during the execution of these tasks and use these resources for executing additional replicas of the most critical tasks in $\mathcal{T}_{active}$. In order for this, first, for a given task graph we compute its concurrency graph, which determines which set of tasks is likely to be executed concurrently. Once a set of tasks is being processed, we may use the concurrency graph to obtain the set of all possibly concurrent tasks, according to which the number of idle resources can be found. Finally, we determine the most critical tasks and schedule extra replicas of those tasks on idle resources.

Once a replica of a task is ready for submission, another important problem is the issue of resource selection. Existing algorithms, such as those based on the HEFT algorithm [14], select the resource that is

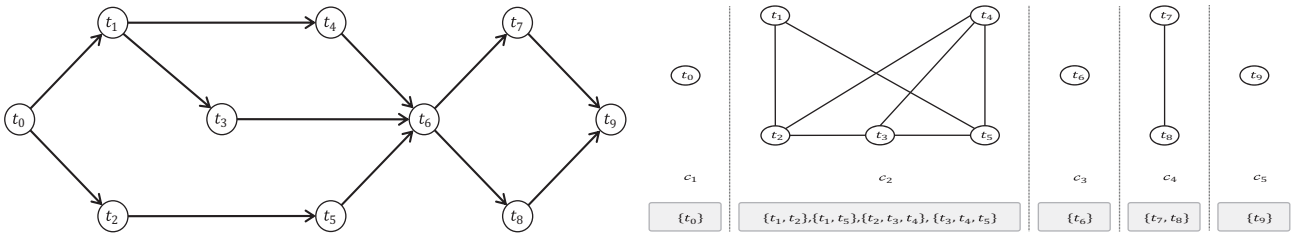
likely to finish the task faster. However, these works do not consider the fluctuations in processing capacities of resources. In our work, the appropriate resource is chosen according to the instantaneous processing capacities of resources. In the following, in Section 4.1 we explain the notions of concurrency graph and concurrency cliques. Then, in Section 4.2, the redundancy selection algorithm is explained. Finally, in Section 4.3, we describe the proposed resource selection method.

4.1. Concurrency graph

While a set $\mathcal{T}_{active}$ of tasks is being executed, in order to determine the number of resources that remain idle during the execution of the tasks in $\mathcal{T}_{active}$, we need to know which set of tasks is likely to be executed in concurrency with the tasks in $\mathcal{T}_{active}$. With this aim, we introduce the notion of the concurrency graph as follows:

Definition 2 *(Concurrency graph). Considering a task graph $\mathcal{W} = (\mathcal{T}, \mathcal{D})$ with the set $\mathcal{T} = \{t_1, \ldots, t_n\}$ of tasks and the set $\mathcal{D}$ of dependencies, the concurrency graph of $\mathcal{W}$ is a graph $\mathcal{P} = (\mathcal{T}, \mathcal{C} = \overline{\mathcal{D}^* \cup \mathcal{D}^{*^T}})$ where $\mathcal{D}^*$ is the transitive closure of $\mathcal{D}$ and $\mathcal{R}^T$ and $\overline{\mathcal{R}}$ are respectively transpose and complement of relation $\mathcal{R}$.*

Given two tasks t_i and t_j, we have $(t_i, t_j) \in \mathcal{C}$ if and only if t_i and t_j are likely to be executed concurrently. In fact, we have $(t_i, t_j) \in C$ if and only if neither t_i is a prerequisite (not necessarily direct) of t_j nor t_j is a prerequisite of t_i. In Figure 2a, an example of a task graph with 10 tasks is shown. The concurrency graph of this task graph is shown in Figure 2b.



(a) (b)

Figure 2. (a) An example of a task graph and (b) its correspoding concurrency graph.

In order to avoid excessive usage of the replication technique, we try to apportion the set of resources among concurrently executing tasks. Therefore, we need to compute the set of cliques in the concurrency graph. A clique $q = \{t_{i,1}, \ldots, t_{i,s}\}$ represents a set of tasks that are likely to be executed concurrently. The set of maximal cliques in a graph can be computed using the Bron–Kerbosch algorithm [25]. For example, the set of maximal cliques in the concurrency graph in Figure 2b is given below the figure.

4.2. Replication vector-finding algorithm

Consider the problem of scheduling of a task graph $\mathcal{W} = (\mathcal{T}, \mathcal{D})$ with the set $\mathcal{T} = \{t_1, \ldots, t_n\}$ of tasks on m computational resources. The problem of determining the appropriate redundancies for a task graph is the problem of finding the redundancy vector $\mathcal{R}$ whose ith element, $\mathcal{R}(t_i), i = 1, \ldots, n$, denotes the number of replicas of the task t_i. We use the resubmission technique for t_i in the case where we have $\mathcal{R}(t_i) = 1$ and use the replication technique in the case of $\mathcal{R}(t_i) > 1$. The number of replicas of tasks is determined such that a

certain criterion is optimized. In this paper, we focus on minimizing the execution time of the task graph while taking into consideration the execution cost.

In the following, we explain the proposed algorithm for determining the appropriate redundancies. First, the concurrency graph $\mathcal{P}$ of $\mathcal{W}$ is decomposed into its connected components. Assume that by $\mathcal{CC} = \{c_1, \ldots, c_r\}$ we denote the set of connected components of $\mathcal{P}$ and we have $c_k = (\mathcal{T}_k, \mathcal{C}_k), k = 1, \ldots, r$. Consider two connected components c_1 and c_2 of the concurrency graph. Since no task in $\mathcal{T}_1$ may be executed in concurrency with any task in $\mathcal{T}_2$, there will be no contention between tasks in these two connected components. Therefore, decisions on choosing the appropriate redundancies for tasks in these connected components can be made independently. Generally, the problem of finding the appropriate redundancy vector for a task graph can be reduced to the same problem for each of the connected components of its concurrency graph.

Now consider a connected component $c_k \in \mathcal{CC}$ including the set $\mathcal{T}_k = \{t_{k,1}, \ldots, t_{k,n_k}\}$ of n_k tasks and the set $\mathcal{Q}_k$ of maximal cliques. In the following, we explain the proposed algorithm for computing the redundancy vector $\mathcal{R}$ for c_k. In this algorithm, first, we consider one replica for each task $t_i \in \mathcal{T}_k$, i.e. we set $\mathcal{R}(t_i) = 1$. Then we compute the set $\mathcal{T}_{possible}$ of tasks in $\mathcal{T}_k$ whose number of replicas can be increased. For a task $t_v \in \mathcal{T}_k$ we have $t_v \in \mathcal{T}_{possible}$ if and only if

$$\forall q \in \mathcal{Q}_k \bullet \Big(t_v \in q \Rightarrow \sum_{t_j \in q} \mathcal{R}(t_j) \leq m\Big), \tag{3}$$

which means that if for each maximal clique q in c_k containing t_v, the summation of the number of replicas of tasks is less than or equal to the number of resources, m, then t_v is added to $\mathcal{T}_{possible}$. After the set $\mathcal{T}_{possible}$ is computed we increase the number of replicas of the most critical task in $\mathcal{T}_{possible}$. Then the set $\mathcal{T}_{possible}$ is recomputed and the same process is repeated until $\mathcal{T}_{possible}$ becomes empty.

The proposed redundancy selection algorithm is shown in Figure 3. In the first line of the algorithm, the concurrency graph is computed according to Section 4.1. In line 2, the obtained concurrency graph is decomposed into its connected components. In each iteration of the "for" loop in lines 3 through 13, a connected component $c_k = (\mathcal{T}_k, \mathcal{C}_k)$ of the concurrency graph is processed independently. First, in lines 4 through 6, one replica is considered for each task in c_k. Then in line 7, the set $\mathcal{T}_{possible} \subseteq \mathcal{T}_k$ of tasks whose number of replicas can be increased is computed according to the condition in Eq. (3). Finally, in the "while" loop in lines 8 through 12, while the number of replicas of some task(s) can be increased we find the most critical task t_i among these tasks (line 9) according to Section 4.3. After increasing $\mathcal{R}(t_i)$ in line 10, we recompute $\mathcal{T}_{possible}$ in line 11. The process is repeated until $\mathcal{T}_{possible}$ becomes empty.

4.3. Task criticality metric

As mentioned before, we need to use an appropriate gain metric to indicate how much each task benefits from an extra replica. First, we need to introduce the notion of extended upward rank.

Definition 3 *(Extended upward rank). Assume k_i replicas of a task t_i are being executed concurrently on k_i identical resources. $EU(t_i, k_i)$, the expected upward rank of t_i, is defined as $EU(t_i, k_i) = u(t_i) - T_i + \mathcal{E}(t_i, k_i)$ where $u(t_i)$ and T_i are respectively the upward rank [14] and processing time of t_i and $\mathcal{E}(t_i, k_i)$ is the average executing time of t_i in the presence of failures.*

Algorithm. Redundancy Selection Algorithm

Input: Task graph $\mathcal{W} = (\mathcal{T}, \mathcal{D})$, Number of resources m

Output: Redundancy Vector, $\mathcal{R}$

1. $\mathcal{P} = \left(\mathcal{T}, \mathcal{C} = \overline{\mathcal{D}^* \cup \mathcal{D}^{*T}}\right)$
2. $\mathcal{CC}$ = Compute the connected components of $\mathcal{P}$
3. for each connected component $c_k = (\mathcal{T}_k, \mathcal{C}_k) \in \mathcal{CC}$ do
4. for each task t_i in c_k
5. $\mathcal{R}(t_i) = 1$
6. end
7. $\mathcal{T}_{possible} = \left\{t_v \in \mathcal{T}_k \middle| \forall q \in \mathcal{Q}_k \left(t_v \in q \Rightarrow \quad _{t_j \in q} \mathcal{R}(t_j) < m\right)\right\}$
8. while ($\mathcal{T}_{possible} \neq \emptyset$)
9. Choose the most critical task t_i from $\mathcal{T}_{possible}$ according to Section 4.3
10. $\mathcal{R}(t_i) = \mathcal{R}(t_i) + 1$
11. $\mathcal{T}_{possible} = \left\{t_v \in \mathcal{T}_k \middle| \forall q \in \mathcal{Q}_k \left(t_v \in q \Rightarrow \quad _{t_j \in q} \mathcal{R}(t_j) < m\right)\right\}$
12. end
13. end
14. return $\mathcal{R}$

Figure 3. The proposed redundancy selection algorithm.

For computing $\mathcal{E}(t_i, k_i)$, first we need to compute the average execution time of t_i on a resource with failure rate λ. The probability of successful termination in each execution trial is $e^{-\lambda T_i}$. If k_i replicas of the task are being executed concurrently, the probability of successful termination of at least one replica is $1 - (1 - e^{-\lambda T_i})^{k_i}$. Therefore, since the number of trials until success follows a geometric distribution, if k_i replicas of t_i are being executed concurrently the average execution time of t_i is estimated as follows:

$$\mathcal{E}(t_i, k_i) = \frac{T_i}{1 - (1 - e^{-\lambda T_i})^{k_i}}. \tag{4}$$

Now, for a task t_i with k_i replicas, $\mathcal{G}(t_i, k_i)$, the amount of decrease in extended upward rank of t_i resulted from an increase in the number of replicas of t_i, is computed as $\mathcal{G}(t_i, k_i) = EU(t_i, k_i) - EU(t_i, k_i + 1)$. Also, $\mathcal{G}^*(t_i, k_i)$, the amount of decrease in maximum extended upward rank of all tasks in $\mathcal{T}_{possible}$, is computed as follows:

$$\mathcal{G}^*(t_i, k_i) = \max_{t_j \in \mathcal{T}_{possible}} EU(t_j, k_j) - max\big(EU(t_i, k_i + 1), \max_{t_j \in (\mathcal{T}_{possible} \setminus \{t_i\})} EU(t_j, k_j)\big). \tag{5}$$

Now, the gain value of t_i, denoted as $gain(t_i, k_i)$, is defined as the average of $\mathcal{G}(t_i, k_i)$ and $\mathcal{G}^*(t_i, k_i)$, i.e. we have $gain(t_i, k_i) = 0.5(\mathcal{G}(t_i, k_i) + \mathcal{G}^*(t_i, k_i))$. The gain value of t_i indicates how much t_i benefits from an extra replica.

4.4. Example

For illustration of the proposed redundancy selection algorithm, we apply the proposed algorithm on the task graph in Figure 2a, whose corresponding concurrency graph is shown in Figure 2b. Assume $m = 4$ computational resources are provisioned for scheduling this task graph. For the sake of brevity, we apply the algorithm on connected component c_2 with the set $\mathcal{T}_2 = \{t_1, \ldots, t_5\}$ of tasks. The set $\mathcal{Q}_2$ of maximal concurrency cliques in this component is as follows:

$$\mathcal{Q}_2 = \{q_1 = \{t_1, t_2\}, q_2 = \{t_1, t_5\}, q_3 = \{t_2, t_3, t_4\}, q_4 = \{t_3, t_4, t_5\}\}. \tag{6}$$

Initially, for each task $t_i, i = 1, \ldots, 5$ we let $\mathcal{R}(t_i) = 1$; thus, we have $\mathcal{T}_{possible} = \mathcal{T}_2$. Then, in each round of the algorithm, among the tasks in $\mathcal{T}_{possible}$, the number of replicas of the task with highest gain is increased. Assuming that t_3 is the task with the maximum gain, it is the first task whose number of replicas is increased. Thus, we let $\mathcal{R}(t_3) = 2$ and the gain value of t_3 is reduced to $gain(t_3, 2)$. Now, assume t_2 is the most critical task. After increasing $\mathcal{R}(t_2)$ the summation of the number of replicas of tasks in q_3 reaches $m = 4$ and tasks t_2, t_3, and t_4 are removed from $\mathcal{T}_{possible}$. Thus, we have $\mathcal{T}_{possible} = \{t_1, t_5\}$. Assume that t_1 is the more critical one of these two tasks, and after increasing $\mathcal{R}(t_1)$ it still has more criticality than t_5. After increasing $\mathcal{R}(t_1)$ once again, as the summation of number of replicas of tasks in q_1 reaches m, t_1 is removed from $\mathcal{T}_{possible}$. Now t_5 is the only task in $\mathcal{T}_{possible}$, $\mathcal{R}(t_1)$ is increased twice, and we have $\mathcal{R}(t_1) = 3$. Finally, t_1 is removed from $\mathcal{T}_{possible}$ and the algorithm is stopped since $\mathcal{T}_{possible}$ is empty. We have $\mathcal{R}(t_1) = 3, \mathcal{R}(t_2) = 2, \mathcal{R}(t_3) = 3, \mathcal{R}(t_4) = 2, \mathcal{R}(t_5) = 3$.

4.5. Resource selection method

Once a replica of a task is ready for submission, another concern is selecting the appropriate resource for processing the replica. In many of the existing works, such as those that are based on [14], a ready task is assigned to the resource that is likely to finish it faster. In finding this resource, we have to take into consideration the size of the task, the available processing capacity of resources, and the amount of work that needs to be processed before the execution of the task can be started. Uncertainty in each of these three items can cause challenges in recourse selection. Existing works assume an environment with fixed processing capacity, i.e. they do not consider the fluctuations in processing capacities of cloud resources. In the following, we propose a method for selecting the resource that is likely to finish the task faster with respect to its instantaneous processing capacity.

In order to obtain an estimation of the instantaneous processing capacities of resources, once an instance of a task t_i is finished at a resource r_i, $ET(t_i, r_j)$, the time required for execution of t_i on r_i is used for estimating the current processing capacity of r_i, which is denoted by $P_{new}(r_i)$. We have $P_{new}(r_i) = Size(t_i)/ET(t_i, r_j)$, where $Size(t_i)$ is the size of t_i. In order to add some momentum, to reduce the adverse impact of sudden changes in processing capacity of the resource, $\mathcal{PC}(r_i)$, current processing capacity of r_i is computed as follows:

$$\mathcal{PC}(s_i) = \beta \times P_{new}(s_i) + (1 - \beta) \times \mathcal{PC}_{old}(s_i), \tag{7}$$

where the momentum parameter, β, takes a value in [0,1] and $\mathcal{PC}_{old}(r_i)$ is our previous estimation of $\mathcal{PC}(r_i)$. This means that each time a new value is obtained for $P_{new}(r_i)$, the new value of $\mathcal{PC}(r_i)$ is computed as a weighted average of the previous value and $P_{new}(r_i)$. For the first usage, when there is no value for $\mathcal{PC}_{old}(r_i)$ and $P_{new}(r_i)$, the value of $\mathcal{PC}(r_i)$ is determined according to the default processing capacity of r_i, which is

obtained through analysis of historical data. Once a response for an instance of a task is received we update our estimation of $\mathcal{PC}(r_i)$ using Eq. (7). In the case of $\beta = 0$, the default processing capacity of r_i is always considered as its available processing capacity.

In addition to the instantaneous available processing capacity of resources, in order to determine the appropriate resource we have to keep track of the amount of work that is waiting to be processed before the task can be started. Therefore, when a replica of a task is submitted to a resource r_i, we update the amount of work in r_i that is waiting to be processed. In determining the resource that is likely to finish a task faster, both the amount of waiting work and the task size are considered.

5. Evaluation and comparison

In this section, we assess the performance of the CGOR algorithm through a simulation study. We implement a simulated environment coded in Java, which simulates task submissions, file transfers and failure, and cost models and considers contention for computational and network resources. The performance fluctuation of computational resources in the simulated environment is adjusted according to the MMPP model. For the purpose of generating random workload, we also implement a synthetic task graph generation procedure similar to the one in [1]. The performance of the CGOR algorithm is compared with some relevant works. The work most relevant to our work is the one proposed in [2]. However, comparing the results against the mentioned work is not fair since it is not designed to consider the possibility of task failures.

The CGOR algorithm is compared against different configurations of the replication method with 2 and 3 replicas for all tasks, respectively denoted by $Rep(2)$ and $Rep(3)$, as well as the resubmission impact (RI) heuristic [6], as it also uses a hybrid of replication and resubmission redundancies. The number of replicas of t_i in the RI heuristic is computed by the formula $\lfloor \sqrt[c]{RI(t_i)} \times rep_{max} \rfloor$. We denote by $RI(rep_{max}, c)$ the resubmission impact heuristic with parameters rep_{max} and c. For each scenario, we apply different configurations of the RI heuristic with different parameter values: $RI(2,1)$, $RI(2,2)$, $RI(3,1)$, and $RI(3,2)$. We denote by $RI(*,*)$ and $Rep(*)$ the best values among these family of respectively RI and Rep configurations. In the following, first we discuss the appropriate values for the parameters of the performance fluctuation model and then we assess the impact of different parameters on the performance of the algorithms.

5.1. Fluctuation model parameters

In [2], resource performance loss was sampled from a normal distribution with average of 15% loss and standard deviation of 10% loss. Similarly, loss in data transfer performance is modeled by a normal distribution with average of 30% loss and standard deviation of 15% loss. In this study, in order to model resource performance variations, we use the MMPP model with a CTMC as shown in Figure 1. In this model, we have

$$\delta_{i,i+1} = \delta, i = 0, \ldots, k-2, \qquad \delta_{i,i-1} = \delta, i = 1, \ldots, k-1. \tag{8}$$

For the processing capacity of resources in state c_i, denoted as C_i, we have $C_i = C_0(1 - i \times \theta), i = 0, \ldots, k-1$ where C_0 denotes the resource processing capacity without any degradation. Therefore, according the expected value and variance formulas for the uniform distribution, the average, C_{avg}, and variance, C_{var}, for processing capacity are computed as follows:

$$C_{avg} = C_0\Big(1 - \frac{k-1}{2}\theta\Big), C_{var} = C_0^2\Big(\frac{k^2-1}{12}\theta^2\Big). \tag{9}$$

In order to obtain the same average performance degradation as in [2], the values of the parameters θ and k are chosen such that we have $C_{avg} = (1 - 0.3)C_0 = 0.7C_0$ and $C_{var} = 0.15^2C_0^2 = 0.225C_0^2$. Therefore, we have $k = 7$ and $\theta = 0.05$, which means that the maximum performance degradation in our performance fluctuation model is 30%.

5.2. Comparison with previous works

In this section, we present a comparative evaluation of the CGOR algorithm and some relevant works. We describe the experiments we conducted in order to evaluate the CGOR algorithm. In our comparisons we consider average execution time, which is defined as the average time required for executing the task graph, and average execution cost. In order to study the performance of the CGOR approach, we have assessed the impact of the following parameters on the above-mentioned criteria: application size, parallelism level, number of resources, task size variance, and resource failure rate. We simulate the scheduling of tasks graphs with $n \in \{200, 400, 600, 800\}$ tasks in an environment with $m = 5$ computational resources. The time to failure of resources is assumed to follow an exponential distribution with average rate $\lambda_{avg} = 0.75 \times 10^{-3}$ s. Finally, the number of simulation runs for each configuration is set to 50.

5.2.1. Application size

In order to assess the impact of varying the application size (the number of task graph nodes) on the performance of the algorithms, the algorithms are executed on task graphs with 200, 400, 600, and 800 tasks. Figure 4a and Figure 4b show the impact of application size on the average values for execution time and cost, respectively. The results show that as the application size increases the CGOR algorithm obtains a more significant improvement over the existing algorithms in terms of both execution time and cost. Also, in Figure 5a, the CDF distribution of the execution time of the algorithms is given for a task graph with $n = 500$ nodes. The results show a significant performance improvement in terms of average execution time.

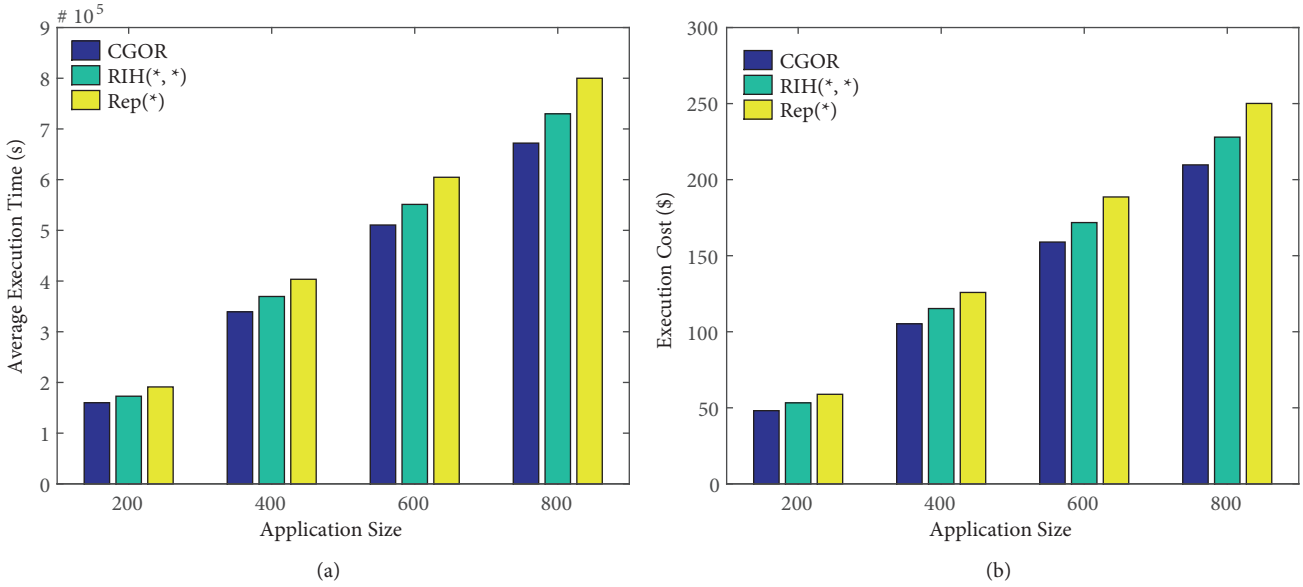


Figure 4. The impact of application size on (a) the average execution time and (b) the average execution cost for different scheduling algorithms.

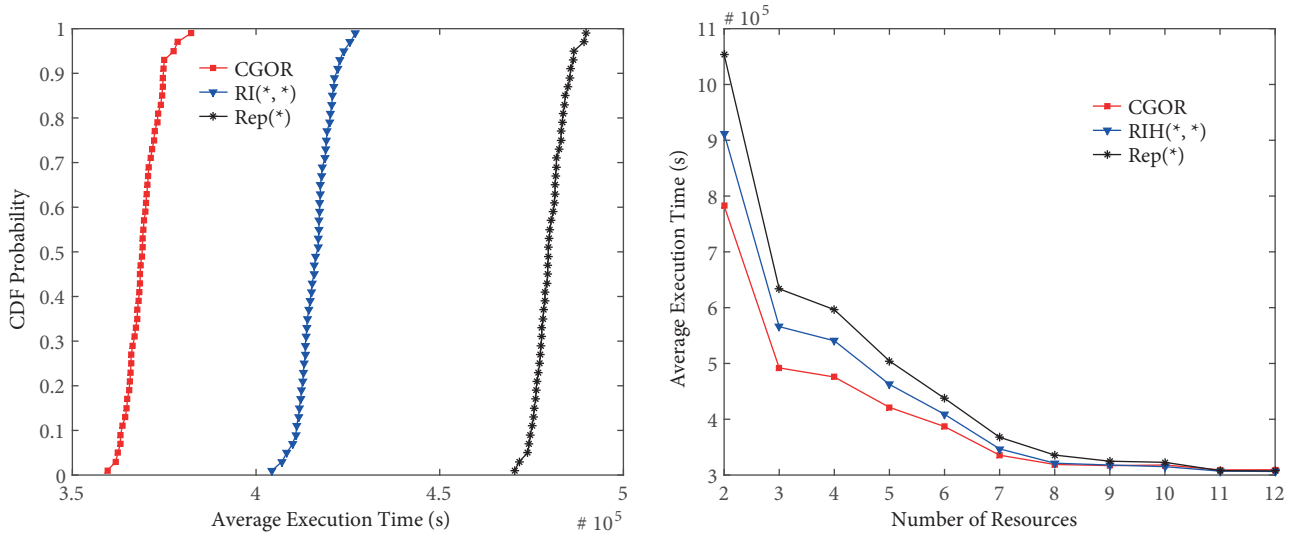


Figure 5. (a) The CDF of average execution times of algorithms; (b) the impact of number of resources on average execution time.

5.2.2. Number of resources and concurrency level

As more resources are provisioned for the execution of the task graph, more tasks can be replicated. Figure 5b shows the average execution time of a task graph with $n = 500$ nodes for environments with different numbers of resources. The results show that in situations with resource shortage the CGOR algorithm achieves a more significant improvement over the prior algorithms. In such situations, the CGOR algorithm is able to adjust the number of replicas of tasks according to the number of available resources. Also, in situations where enough resources are available, the proposed algorithm is tuned to employ more resources for replicating tasks. As the figure shows, increasing the number of resources beyond six is not a good idea as we achieve only a negligible improvement in execution time at the price of a significant cost increase.

The concurrency level of the graph, in the random task graph generation procedure, is controlled by a parameter W representing the maximum graph width. Therefore, in order to assess the impact of graph concurrency level experiments are repeated for different values of W from five to twelve. The results in Figure 6a show a performance improvement in almost all cases.

5.2.3. Average resource failure rate

In order to assess the impact of varying failure rate on the performance of algorithms, the average failure rate is varied from 0.0075×10^{-5} to 0.7200×10^{-5} s. Figure 6b shows the impact of failure rate on average execution time of scheduling algorithms.

5.2.4. Average data and task size

The impact of input/output data size is assessed by varying the maximum data size from 0.1×10^6 KB to 1.9×10^6 KB. The results, in Figure 7a, show that performance improvement of the proposed approach is more significant for applications with higher computation to communication (CCR) ratio. This is because in choosing resources only task sizes are taken into consideration. Also, Figure 7b shows the impact of varying average task

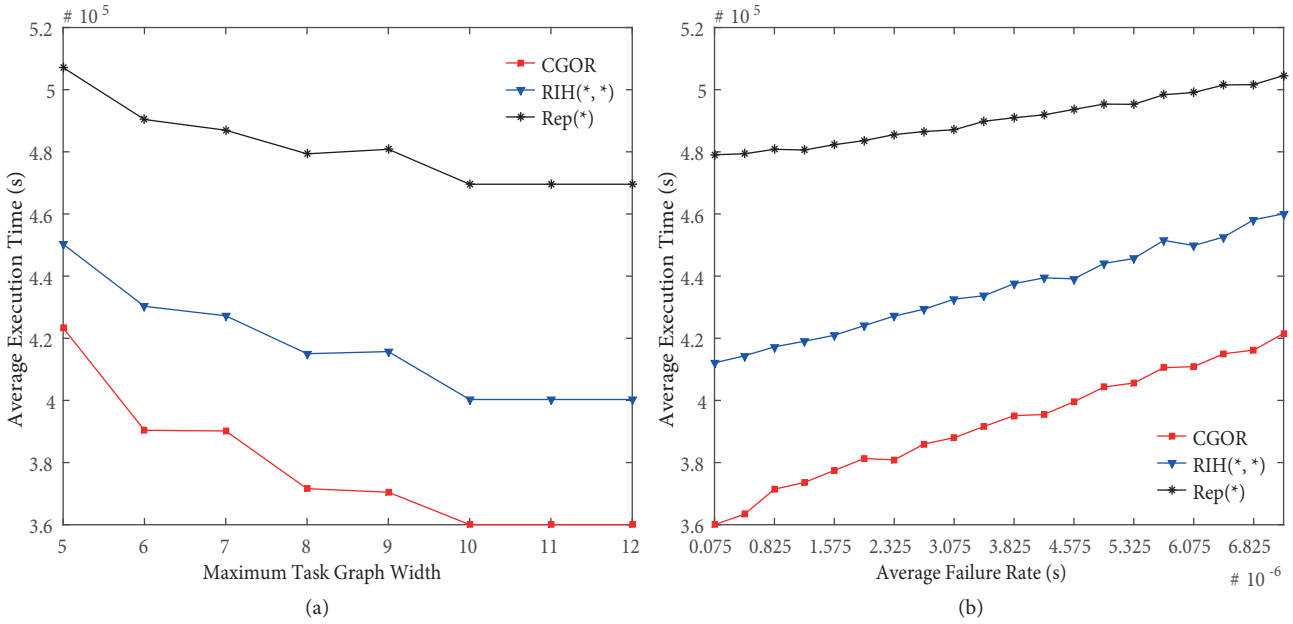


Figure 6. The impact of (a) number of resources and (b) average failure rate on average execution time.

size. In the random graph generation procedure, tasks sizes are randomly chosen from a uniform distribution with a minimum of 5×10^6. In order to obtain different average values, experiments are repeated with different values for maximum task size from 0.1×10^8 to 3.1×10^8. The results show that the amount of improvement is more significant for task graphs with larger tasks.

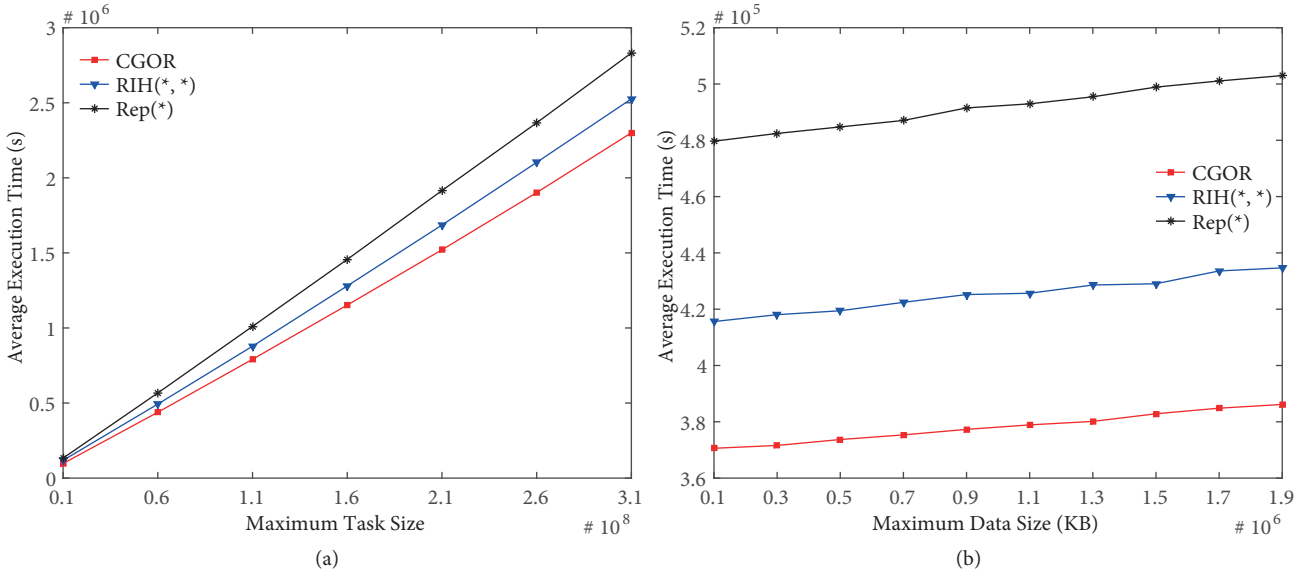


Figure 7. The impact of (a) average task size and (b) average data size on execution time.

5.2.5. Fluctuation model

In order to assess the impact of fluctuation model behavior, we examine the performance of the algorithms in environments with different fluctuation rates and different levels of performance degradation. In order to assess the impact of the rate of fluctuations we have simulated the scheduling of a task graph with $n = 500$ nodes for environments with different performance fluctuation rates from 0.1×10^{-4} to 10×10^{-4}. Figure 8a shows the impact of varying fluctuation rate on the relative performance of the algorithms. The results show that all algorithms have better performance when the fluctuation rate is low. The relative performance of the algorithms is the same for all fluctuation rates. Figure 8b shows the impact of maximum performance degradation on the performance of the algorithms. The amount of performance degradation is varied from 10% to 70%. It is worth mentioning that we consider performance fluctuations only for computational resources. The results show the performance improvement of the CGOR algorithm in all cases.

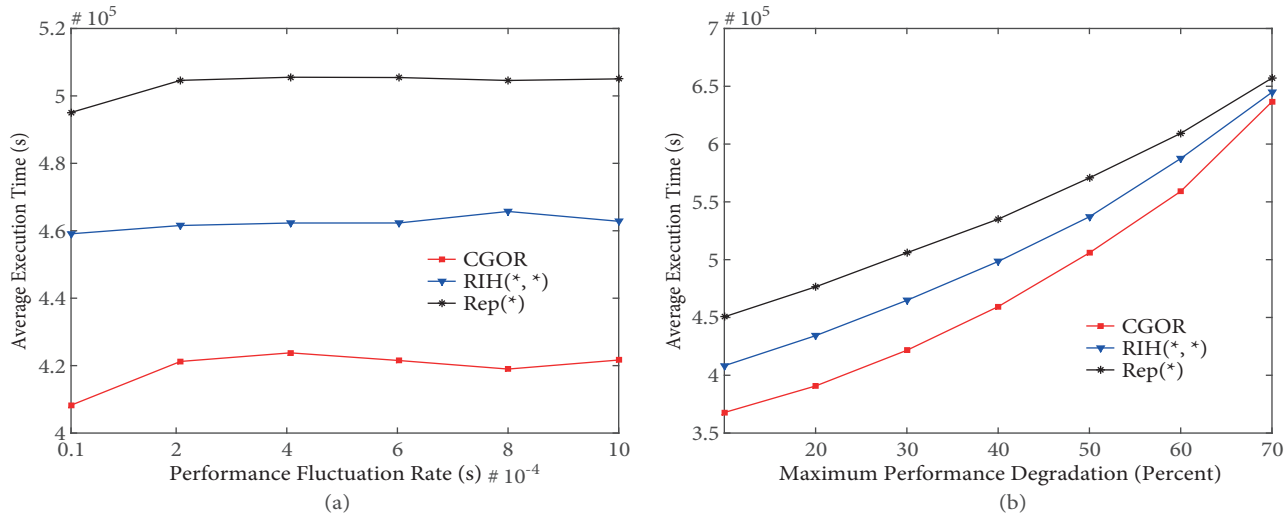


Figure 8. The impact of (a) average performance fluctuation rate and (b) maximum performance on execution time.

6. Conclusions and future work

In this paper, we presented a new algorithm, called CGOR, for scheduling task graphs in environments with failure-prone and performance-varying resources. Technical contributions of the CGOR algorithm are as follows: 1) Based on the introduced notion of concurrency graphs, we have proposed a structure-aware approach for finding idle resources and use them for bringing more resilience. Unlike the work in [2], the proposed algorithm does take into consideration the possibility of task failures. 2) We propose a new redundancy selection algorithm that determines the optimal combination of the two redundancy techniques according to the number resources with idle durations and the criticalities of active tasks. In order to avoid the drawbacks of the resubmission technique, the CGOR algorithm tries to use idle resources for task replication. 3) We propose a resource selection criterion for selecting appropriate resources with respect to their instantaneous processing capacities. We use a MMPP-based performance fluctuation model for modeling the changes in the performance of computational resources. The performance of the CGOR approach is assessed using a simulation study and is compared against the prior state-of-the-art algorithms. We develop a random task graph generation procedure for generating synthetic workloads. Based on the experimental study, using a large set of randomly generated task graphs,

the CGOR algorithm significantly outperformed the previous algorithms in terms of both performance and cost metrics in most situations.

In the future, we are going to include the checkpointing redundancy in the proposed hybrid redundancy selection algorithm.

References

[1] Chandrashekar DP. Robust and fault-tolerant scheduling for scientific workflows in cloud computing environments. PhD, University of Melbourne, Melbourne, Australia, 2015.

[2] Calheiros RN, Buyya R. Meeting deadlines of scientific workflows in public clouds with tasks replication. IEEE T Parall Distr 2014; 25: 1787-1796.

[3] Pacheco-Sanchez S, Casale G, Scotney B, McClean S, Parr G, Dawson S. Markovian workload characterization for QoS prediction in the cloud. In: 2011 IEEE International Conference on CLOUD; 4–9 July 2011; Washington, DC, USA. pp. 147-154.

[4] Abrishami S, Naghibzadeh M, Epema DHJ. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. Future Gener Comp Sy 2013; 29: 158-169.

[5] Jackson KR, Ramakrishnan L, Muriki K, Canon S, Cholia S, Shalf J, Wasserman HJ, Wright NJ. Performance analysis of high performance computing applications on the Amazon web services cloud. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science; 30 November–3 December 2010; Indianapolis, IN, USA. pp. 159-168.

[6] Plankensteiner K, Prodan R. Meeting soft deadlines in scientific workflows using resubmission impact. IEEE T Parall Distr 2012; 23: 890-901.

[7] Benoit A, Hakem M, Robert Y. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In: IEEE International Symposium on Parallel and Distributed Processing; 14–18 April 2008; Miami, FL, USA. pp. 1-8.

[8] 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 2009; 69: 282–294.

[9] Das A, Sarkar AD. On fault tolerance of resources in computational grids. International Journal of Grid Computing and Applications 2012; 3: 1-10.

[10] Aupy G, Benoit A, Casanova H, Robert Y. Checkpointing strategies for scheduling computational workflows. International Journal of Networking and Computing 2016; 6: 2-26.

[11] Fischer W, Meier-Hellstern K. The Markov-modulated Poisson process (MMPP) cookbook. Perform Evaluation 1993; 18: 149-171.

[12] Gu Y, Wu C, Liu X, Yu D. Distributed throughput optimization for large-scale scientific workflows under fault-tolerance constraint. Grid Computing 2013; 11: 361-379.

[13] Girault A, Kalla H, Sighireanu M, Sorel Y. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In: 2003 International Conference on Dependable Systems and Networks; 22–25 June 2003; San Francisco, CA, USA. pp. 159-168.

[14] Topcuoglu H, Hariri S, Wu M. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE T Parall Distr 2002; 13: 260-274.

[15] Lin C, Lu S. SCPOR: An elastic workflow scheduling algorithm for services computing. In: IEEE International Conference on Service-Oriented Computing and Applications; 12–14 December 2011; Irvine, CA, USA. pp. 1-8.

[16] Ang TF, Ling TC, Phang KK. Adaptive QoS scheduling in a service-oriented grid environment. Turk J Electr Eng Co 2012; 20: 413-424.

[17] Sajid M, Raza Z. Energy-aware stochastic scheduling model with precedence constraints on DVFS-enabled processors. Turk J Electr Eng Co 2016; 24: 4117-4128.

[18] Bougeret M, Casanova H, Rabie M, Robert Y, Vivien F. Checkpointing strategies for parallel jobs. In: Conference on High Performance Computing Networking, Storage and Analysis; 12–18 November 2011; Seattle, WA, USA. pp. 33:1-33:11.

[19] Benoit A, Cavelan A, Robert Y, Sun H. Two-level checkpointing and verifications for linear task graphs. In: IEEE International Parallel and Distributed Processing Symposium Workshops; 23–27 May 2016; Chicago, IL, USA. pp. 1239-1248.

[20] Zhang Y, Mandal A, Koelbel C, Cooper KD. Combined fault tolerance and scheduling techniques for workflow applications on computational grids. In: IEEE/ACM International Symposium on Cluster Computing and the Grid; 18–21 May 2009; Shanghai, China. pp. 244-251.

[21] Rajabi A, Wong JW. MMPP characterization of web application traffic. In: IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems; 7–9 August 2012; Washington, DC, USA. pp. 107–114.

[22] Rajabi A, Wong JW. Provisioning of computing resources for web applications under time-varying traffic. In: IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems; 9–11 September 2014; Paris, France. pp. 152–157.

[23] Ryden T. An EM algorithm for estimation in Markov-modulated Poisson processes. Comp Stat Data Anal 1996; 21: 431-447.

[24] Dong F. Copula theory and its applications in computer networks, PhD, University of Victoria, Victoria, Canada, 2017.

[25] Eblen JD, Phillips CA, Rogers GL, Langston MA. The maximum clique enumeration problem: algorithms, applications and implementations. Lect Notes Bioinformat 2011; 13: 306-319.