Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

# EDQWS: an enhanced divide and conquer algorithm for workflow scheduling in cloud

Ghazaleh Khojasteh Toussi[1], Mahmoud Naghibzadeh[1*], Saeid Abrishami[1], Hoda Taheri[1] and Hamid Abrishami[2]

**Abstract**
A workflow is an effective way for modeling complex applications and serves as a means for scientists and researchers to better understand the details of applications. Cloud computing enables the running of workflow applications on many types of computational resources which become available on-demand. As one of the most important aspects of cloud computing, workflow scheduling needs to be performed efficiently to optimize resources. Due to the existence of various resource types at different prices, workflow scheduling has evolved into an even more challenging problem on cloud computing. The present paper proposes a workflow scheduling algorithm in the cloud to minimize the execution cost of the deadline-constrained workflow. The proposed method, EDQWS, extends the current authors' previous study (DQWS) and is a two-step scheduler based on divide and conquer. In the first step, the workflow is divided into sub-workflows by defining, scheduling, and removing a critical path from the workflow, similar to DQWS. The process continues until only chain-structured sub-workflows, called linear graphs, remain. In the second step which is linear graph scheduling, a new merging algorithm is proposed that combines the resulting linear graphs so as to reduce the number of used instances and minimize the overall execution cost. In addition, the current work introduces a scoring function to select the most efficient instances for scheduling the linear graphs. Experiments show that EDQWS outperforms its competitors, both in terms of minimizing the monetary costs of executing scheduled workflows and meeting user-defined deadlines. Furthermore, in more than 50% of the examined workflow samples, EDQWS succeeds in reducing the number of resource instances compared to the previously introduced DQWS method.

**Keywords:** Workflow scheduling, Cloud computing, Critical path, Merging algorithm, Divide and conquer, Scoring function

## Introduction

Nowadays, cloud computing offers an opportunity to execute scientific applications composed of hundreds or thousands of interrelated tasks [1]. In this context, the workflow model is an effective way to construct such complex applications. It consists of application tasks specified by nodes and the connection lines among nodes which create dependencies among these tasks in a directed acyclic graph (DAG) [2]. The workflow scheduling problem in the cloud aims to assign the tasks to computing resources in order to preserve task precedence while meeting some performance criteria [3].

Besides the total execution time of workflows, most research on workflow scheduling in the cloud has focused on optimizing the total usage cost of computing resources offered by cloud providers [2]. Moreover, faster and more powerful computing resources in the cloud are usually more expensive than slower ones. Therefore, the execution cost can be affected by employing powerful computing resources as this decreases the workflow execution time. Thus, the trade-off between cost and time

*Correspondence: naghibzadeh@um.ac.ir
[1] Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran
Full list of author information is available at the end of the article

is a major challenge of workflow scheduling in the cloud [4]. To address this trade-off, two common methods are employed: either minimizing the total execution time under a budget constraint or minimizing the monetary cost under a deadline constraint.

Depending on user requirements and the workflow application, both cost and time can be considered as constraints or optimization objectives. Some workflows are structured in such a way that the results can be used whenever the execution of the workflow is completed before the deadline. The workflows of medical simulations and weather forecasting are examples of deadline-constrained applications [4]. To address such deadline-constrained applications, the current study proposes a new workflow scheduling algorithm focused on minimizing the total execution cost while respecting the user-defined deadline.

The proposed algorithm, named *enhanced divide and conquer workflow scheduling* (EDQWS), is an extension of the present authors' earlier study [5] which introduced a type of a divide and conquer-based approach. Similar to this previous work, in EDQWS, a large workflow is broken into a number of sub-workflows by identifying the critical path of a workflow and then scheduling and removing it. The same series of steps are repeated in all sub-workflows. Creating sub-workflow with a chain structure, called linear graph, is the stop condition of the algorithm. The present authors extend their previous method of scheduling linear graphs. Since the scheduling of linear graphs as solvable problems is performed in the final step, the linear graphs resulting from different stages of the division process are independent of each other and can therefore be separately scheduled. However, the specific order or combination of their scheduling can affect the selected resource types and the total execution cost. To this end, the current study proposes a scoring function for determining the linear graph combination score on different resources. At each stage, the combination with the highest score is selected and the new combined graph is replaced by the original ones. This combination is binary and will continue until the scoring function reaches a negative score for all new combinations. Owing to the large number of linear graphs, the score calculation is performed in parallel. The present study also introduces a new merge list algorithm that combines the tasks of linear graphs with respect to the laxity of the tasks. To evaluate the performance of the proposed method, it is compared with several state-of-the-art scheduling algorithms. The experimental results determine that the presented method outperforms others in total execution cost and the success rate of meeting deadlines. Moreover, the introduced approach can reduce the number of instances required for scheduling when compared to the previous work of the current authors. The rest of the present article is organized as follows: Section 2 reviews related workflow scheduling. Section 3 formulates the scheduling model and describes its workflow application and resource model. Details of the proposed algorithm are provided in Section 4. Section 5 discusses the experimental results. Finally, the conclusion and future works are presented in Section 6.

## Related work

In recent decades, workflow scheduling has been extensively investigated by academia and industrial researchers. For a traditional distributed system, such as a grid and cluster, most existing research in workflow scheduling focuses on how to minimize the workflow execution time. However, workflow scheduling in the cloud environment is mainly a multi-objective problem. Thus, aside from the execution time, various criteria, such as monetary cost, energy usage, reliability, and security, are considered as QoS requirements [6–10]. Among these, monetary cost and execution time are substantial requirements for workflow scheduling algorithms [2, 11–15]. Normally, for multi-objective cases in which some objectives must be optimal, it is difficult to solve the workflow scheduling problem in the cloud due to its NP-completeness [16]. Therefore, various meta-heuristic and heuristic techniques have been adopted to obtain near-optimal solutions. This section briefly reviews several well-known heuristics and meta-heuristic workflow scheduling algorithms related to the present study's proposed method. As the main category, deadline-aware workflow scheduling algorithms are first reviewed. This is followed by a discussion on budget-aware and then multi-objective schedulers.

### Deadline-aware workflow scheduling

Malawski et al. [17] present a mathematical model to optimize the workflow scheduling cost under a deadline constraint. Their method considers a multi-cloud environment and formulates the scheduling problem as a mixed-integer program (MIP). Abrishami et al. [18] utilize the Partial Critical Path (PCP) concept to develop a deadline-constrained workflow scheduler, named IC-PCP, in a cloud environment. IC-PCP [18] aims to minimize the overall execution cost of the workflow by determining a sequence of tasks as the partial critical paths (PCPs) and mapping all of these tasks to the same VM instance. The preference of the algorithm is to utilize the already leased instances which are able to meet the deadline. IC-PCP distributes the overall deadline to the PCPs. The Enhanced IC-PCP with Replication (EIPR) algorithm [19] attempts to further reduce costs by replicating tasks during the idle times of instances and eliminating some communications. Its experimental results show that the probability of meeting deadlines increases via task replications. The Deadline Constrained Critical Path (DCCP) [20] is a list-based scheduling algorithm on the cloud that aims to meet the user-defined deadline while minimizing the overall workflow execution cost. In the preprocessing step, tasks are partitioned into different levels, to each of which a sub-deadline is assigned. These deadlines are distributed non-uniformly among all levels so that the levels with a longer task execution time receive a longer sub-deadline. In the task prioritization step, DCCP utilizes a concept called the Constrained Critical Path (CCP) to assign all tasks on a path to one resource in order to reduce the communication time of the whole workflow. DCCP finds all CCPs and creates a list based on their modified rank. In each step, the ready tasks of each CCP are mapped to an appropriate resource and other tasks remain for the next steps. Rodriguez and Buyya [21] propose a metaheuristic scheduler in the cloud that intends to minimize the execution cost for deadline-constrained workflows. In this algorithm, resource provisioning and task assignment are integrated as a particle swarm optimization problem. The algorithm produces a near-optimal schedule that determines the number and types of VMs with their leasing period and task assignment. Guo et al. [1] also introduce a PSO[1]-based algorithm for scheduling a deadline-constrained workflow across multiple clouds. Their algorithm minimizes the execution cost of the workflow while meeting the user-defined deadline. Furthermore, the algorithm optimizes the performance for both computation cost and data transfer cost across multiple clouds. Proportional Deadline Constrained (PDC) [12] is a workflow scheduling algorithm on the cloud that attempts to meet the user-defined deadline while minimizing the execution cost. In the preprocessing step, PDC partitions tasks into different levels and assigns a sub-deadline to each level. The user-defined deadline is distributed non-uniformly among all levels so that levels with a longer task execution time receive a longer sub-deadline. PDC creates a list of ready tasks and prioritizes them according to a downward rank. In [2], two schedulers, namely L-ACO[2] and ProLiS,[3] are presented to schedule the deadline-constrained workflow application. ProLiS is a list scheduling algorithm that performs deadline distribution based on the new definition of the probabilistic upward rank. L-ACO is a meta-heuristic algorithm that accomplishes the cost optimization of a deadline-constrained workflow based on ant colony optimization. Deadline distribution and service selection in L-ACO is the same as in ProLiS.

[1] Particle swarm optimization (PSO)

[2] List scheduling ant colony optimization (L-ACO)

[3] Deadline-constrained probabilistic list scheduling (ProLiS)

### Budget-aware workflow scheduling

A budget indicates the maximum amount of money that users are willing to pay for the execution of a workflow application in cloud resources [3]. In [22], the authors propose a Heterogeneous Budget Constrained Scheduling (HBCS) algorithm that minimizes the total workflow while meeting the user's specified budget. The HBCS defines an attribute called worthiness which combines the time and cost factors for the current task resource selection. Faragardi et al. [4] introduce Greedy Resource Provisioning and a modified HEFT (GRP-HEFT) for minimizing the workflow execution time subject to a budget constraint. They propose a greedy algorithm to list the instance types according to their efficiency rate and modify the HEFT [23] algorithm so that it considers a budget constraint. In [24], Wu et al. present a heuristic algorithm, called PCP-B, to schedule a workflow with a budget constraint. PCP-B implements the idea of balancing a budget among the partial critical paths according to their parallel or sequential structural nature. Budget distribution is performed based on the binary search method.

### Multi-objective workflow scheduling

In this category, most strategies try to find a suitable mapping of workflow tasks to cloud resources that respects deadline and budget constraints at the same time. Budget and Deadline Constraint Heterogeneous Earliest Finish Time (BDHEFT) [15] is a multi-objective algorithm proposed to schedule workflow applications on a cloud. BDHEFT leverages the upward ranks to assign a priority to each task. In addition, a set of best possible resources is constructed for each selected task via the following six variables: Spare Workflow Budget (SWB), Spare Workflow Deadline (SWD), Current Task Deadline (CTD), Current Task Budget (CTB), Adjustment Factor (BAF), and Deadline Adjustment Factor (DAF). By considering the spare deadline and spare budget of each task, the resource is selected from the best possible resource set for each task, in which the overall execution time and execution cost of the workflow execution are simultaneously minimized. Durillo and Prodan propose the multi-objective heterogeneous earliest finish time (MOHEFT) algorithm [25] as an extension of HEFT [23]. MOHEFT computes a set of Pareto-based solutions from which users can select the best one. As noted by the authors, most of the solutions computing the Pareto-front are based on genetic algorithms. The algorithm is generic in terms of the number and types of objectives and so the makespan and overall cost of the workflow applications can be optimized. Wu et al. [26] present a PSO-based strategy for workflow scheduling in the clouds. Their aim is to reduce either the makespan or cost while satisfying either the budget or deadline constraints. The elasticity of resource provision is ignored and it is assumed that several initialized VMs are available in advance. In [27], a heuristic-based scheduling algorithm is proposed to schedule the workflow under deadline and budget constraints. The algorithm utilizes a novel trade-off factor between time and cost to determine the most viable scheduling and the most appropriate instance type for provisioning.

## Problem statement

The present study addresses the problem of the cost optimization of deadline constrained workflow scheduling in the cloud. This section first explains the workflow model, resource model, and definitions related to this problem. Subsequently, the problem formulation is presented.

### Workflow model

A workflow application can be modeled as a directed acyclic graph (DAG), $G=(V,E)$, where $V=\{t_1,t_2,\ldots,t_n\}$ is a set of all workflow tasks illustrated by graph vertices and $E=\{e_{i,\ j}=(t_i,t_j)|\,t_i,t_j\in V\}$ represents the dependencies among the tasks. Each $e_{i,\,j}$ indicates the precedence between $t_i$ and $t_j$, meaning that $t_j$ can be performed when $t_i$ is completed. Besides the dependencies, the data transmission among tasks is represented by the weight attached to $e_{i,\,j}$ . Furthermore, $data_{i,\,j}$ shows the amount of data transferred to $t_j$ after $t_i$ is completed; hence, the execution of $t_j$ can only start after $data_{i,\,j}$ has already been made available. On the other hand, a task can be executed if all its predecessors are terminated. Note that, on each edge, $e_{i,\,j}$, $t_i$ is a predecessor of $t_j$ and $t_j$ is a successor of $t_i$. Each task may have one or more predecessors and successors except for $t_{entry}$ and $t_{exit}$ . $t_{entry}$ is a task with no predecessor and $t_{exit}$ is a task with no successor. To generalize the workflow with one entry and one exit, two dummy tasks, $t_{entry}$ and $t_{exit}$, with zero execution time and without data transmission, are added to the beginning and the end of the workflow, respectively. Fig. 1 illustrates a sample workflow represented by a DAG.

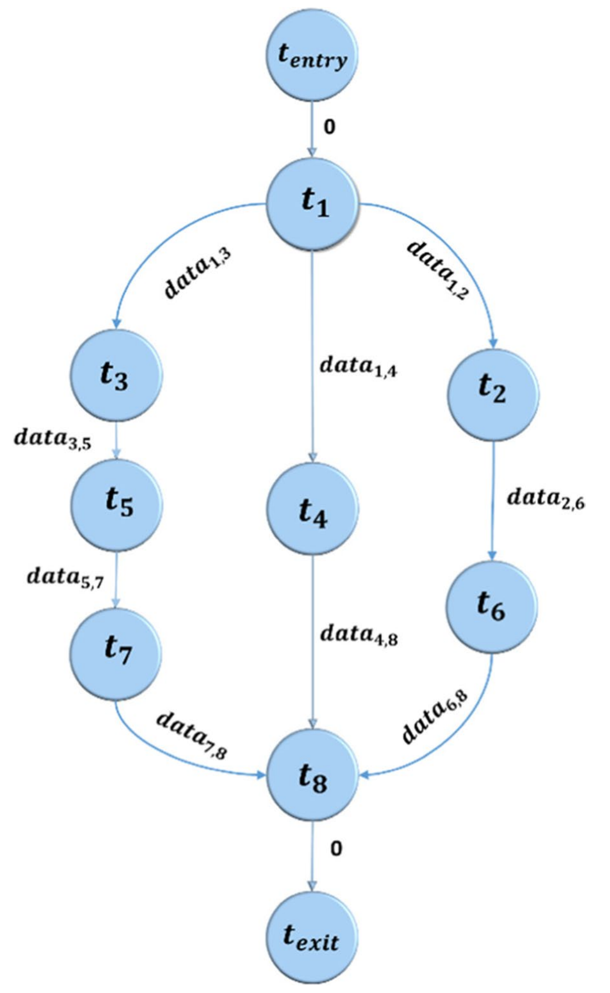


**Fig. 1** Workflow sample

### Resource model

The present article considers IaaS as a cloud service provider. IaaS provides a variety of computational resources with different costs via virtual machines (VMs) that feature different processing capabilities, memory, and storage. VMs with higher processing capabilities are assumed to have higher costs. A running virtual machine is called an instance and users can request infinite instances from the cloud service provider. In the current study, instances are provisioned on-demand and the pricing model is considered to be pay-as-you-go hourly-based, which is widely used by large public cloud providers [4]. In this model, a user must pay for the whole hour even though use of an instance is for less than an hour. The current work denotes $VM=\{vm_1,vm_2,\ldots,vm_n\}$ as a set of heterogeneous computational resources offered by the cloud service provider via VM and $Cost=\{C_1,C_2,\ldots,C_n\}$ as the cost of using each VM for an hour. It is also assumed that all computational resources are in the same region. Thus, the average bandwidth between instances is roughly identical and internal data transfer is free of charge [2].

### Definitions

#### *Task execution time*

Given that the VMs offered by the service provider are heterogeneous, each task has a different execution time based on the type of VM running on it. Therefore, the execution time of task $t_i$ on $vm_j$ is denoted by $w_{i,j}$. It should be noted that $w_{i,j}$ is the worst-case execution time of $t_i$ on $vm_j$ and it is assumed that only one task can be executed on each VM instance at any time.

#### *Communication time*

The communication time between two tasks is the amount of time necessary to transfer data from $t_i$ to $t_j$, as shown in (1). When both tasks are executed on the same instance, the communication time becomes zero.

$$CM_{i,j} = \begin{cases} 0 \; if \; Inst(t_i) = Inst\left(t_j\right) \\ \frac{data_{i,j}}{BW} \; otherwise \end{cases} \tag{1}$$

where $Inst(t_i)$ represents the instance on which $t_i$ is mapped, $data_{i,j}$ shows the amount of data transferred to $t_j$ after $t_i$ 's completion, and $BW$ is the bandwidth between the two instances.

#### *Earliest start time (EST) and earliest finish time (EFT)*

For each unscheduled task, $t_i$, $EST(t_i)$ is defined as the earliest time when $t_i$ can start its execution after all its predecessors have finished and after having received the associated data. EST is calculated as follows:

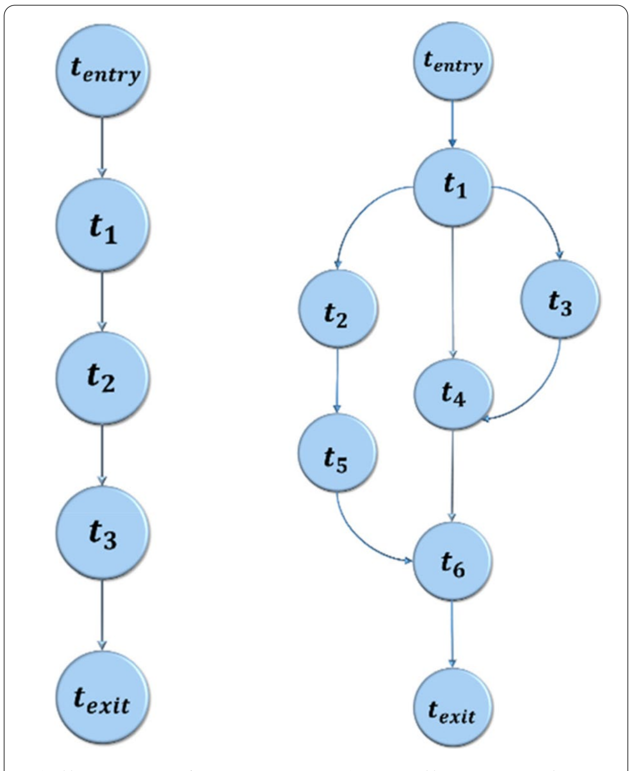


**Fig. 2** Examples of a linear graph and a nonlinear graph

$$EST(t_i) = \begin{cases} 0 \; if \; t_i = t_{entry} \\ \max\limits_{t_j \in predecessor(t_i)} EST\left(t_j\right) + W\left(t_j\right) + CM_{j,i} \; otherwise \end{cases} \tag{2}$$

where $CM_{j,i}$ is the communication time and $W(t_j)$ is the shortest execution time of $t_j$, which is defined as follows:

$$W\left(t_j\right) = \min_{k \in vm \; types} \left\{w_{j,k}\right\} \tag{3}$$

According to the earliest start time definition, the earliest finish time of each unscheduled task, $t_i$, can be calculated as follows:

$$EFT(t_i) = EST(t_i) + W(t_i) \tag{4}$$

#### *Actual start time (AST) and actual finish time (AFT)*

After assigning a task to the desired instance, AST and AFT are obtained for each task. These values can be different from the *EST* and *EFT* that are determined before scheduling. Equation (5) shows the relation between these parameters.

$$AST(t_i) \geq EST(t_i) \tag{5}$$

$$AFT(t_i) = AST(t_i) + w_{i,j}$$

where $j$ is the type of instance to which $t_i$ is assigned.

#### *Critical path and critical tasks*

The critical path (CP) in a workflow is the path from the entry task to the exit task which has the maximum summation of task execution times and inter-task communication times among all paths [23]. All the tasks on a critical path are called critical tasks.

#### *Linear graph and nonlinear graph*

The present paper divides the workflow graphs into two categories: linear graphs and nonlinear graphs. A linear graph is a connected graph with a chain structure. Each vertex in this graph, which corresponds to a task in the workflow, has exactly one parent and one child, except for the entry task, which has no parent, and the exit task, which has no child. If a graph contains other structures, it is called a nonlinear graph. Figure 2 provides a graphic representation of linear and nonlinear graphs.

### Problem formulation

The issue presented in the current article is the deadline-constrained cost optimization of workflow scheduling in an IaaS cloud environment. Considering *wf* as the workflow and *D* as the workflow deadline, this subsection first defines the overall workflow execution cost and then formulates the problem.

The total execution cost of a workflow, $Cost_{total}(wf)$, is the summation of all the costs of the used instances during scheduling, as follows:

$$Cost_{total}\left(wf\right) = \sum_{i=1}^{n} Cost(Inst_i) \tag{6}$$

where $Inst_i$ is the $i^{th}$ leased instance and $n$ is the total number of leased instances. $Cost(Inst_i)$ depends on two parameters: the total time of utilizing $Inst_i$ and the price of $Inst_i$ for an hour. As mentioned earlier, there is an hourly-based price model and the user must pay for the whole hour even though the instance is used for less than an hour. As a result, the instance cost is calculated as follows:

$$Cost(Inst_i) = \left\lceil \frac{\max\limits_{t_j \in Inst_i \ Assigned\ tasks}\left\{AFT\left(t_j\right)\right\} - \min\limits_{t_j \in Inst_i \ Assigned\ tasks}\left\{AST\left(t_j\right)\right\}}{3600} \right\rceil \times C(Inst_i) \tag{7}$$

where $C(Inst_i)$ is the specified cost for utilizing $Inst_i$ for 1 hour. According to the previous definitions, the present study's optimization scheduling problem is applied to achieve a mapping of tasks to suitable instances so as to minimize the total monetary cost while not exceeding the deadline constraint. Relation (8) shows our problem formulation.

$$\text{Minimize } Cost_{total}\ (wf) \tag{8}$$

$$\boldsymbol{subject\ to}\ AFT\left(t_{exit}\right) \le D$$

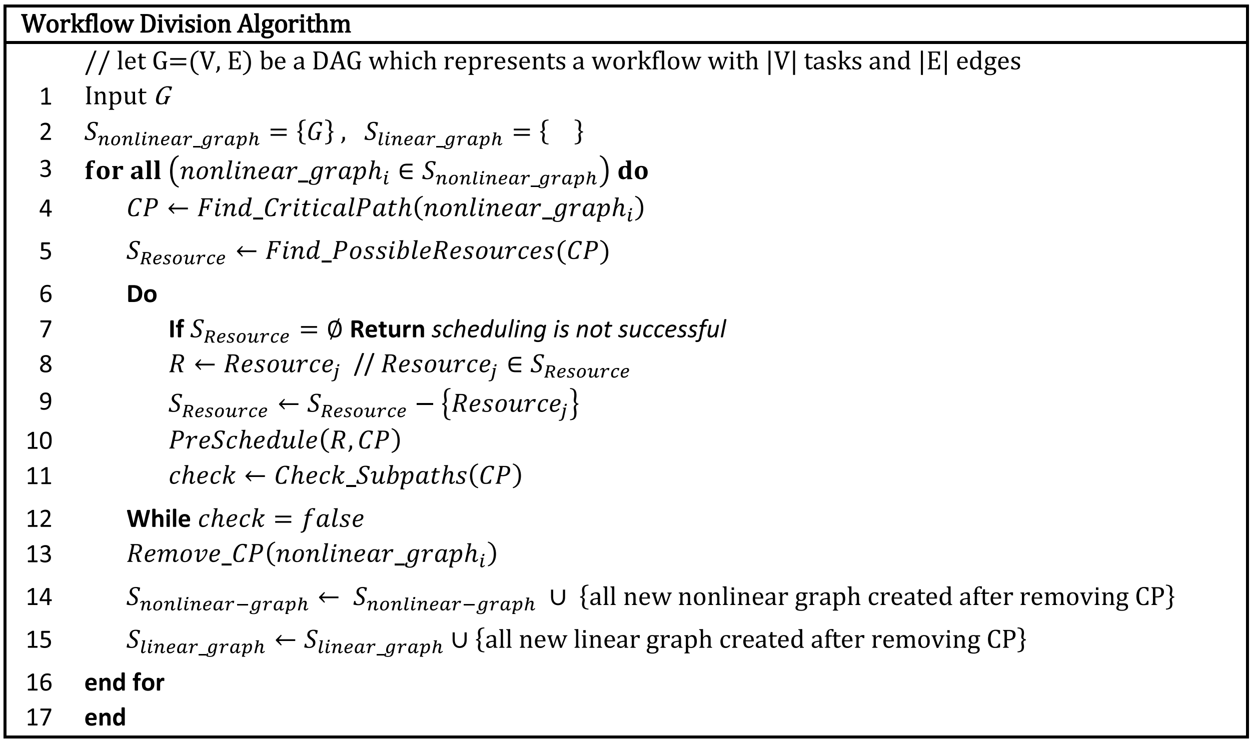
## Proposed method

The proposed solution for the cloud workflow scheduling problem is a static method that aims to minimize the execution cost of the workflow while meeting its deadline. For minimizing the execution cost, the introduced method focuses on the repeated use of the critical path concept. Dividing the initial workflow into several sub-workflows makes it possible to define the critical path in each resulting sub-workflow. According to the workflow structure and the dependencies among tasks, the original divide and conquer method is modified to be consistent with the goals of the current study. The proposed algorithm consists of two main phases: the division phase and the linear graph scheduling phase. In the division phase, the workflow division algorithm is presented to perform the workflow dividing operations according to the current authors' previous research (DQWS) [5]. The division of the workflow is performed by determining, scheduling, and removing the workflow critical path. By removing the critical path, the workflow leftovers are divided into one or more smaller sub-workflows and this process is repeated for each sub-workflow. The creation of sub-workflows as the structure of the linear graphs is the stop condition of the divide and conquer process and the sub-workflow scheduling is considered as a small solvable problem. By scheduling linear graphs with the scoring method, the proposed linear graph scheduling phase takes on a completely different approach from the current authors' previous work. As the difference between the proposed method and the present authors' past work (DQWS) lies in this second phase of the introduced algorithm, Section 4–1 shall first present the generalities of the proposed workflow division algorithm while Section 4–2 will go on to explain the details of the proposed linear graph scheduling algorithm.

### Workflow division algorithm

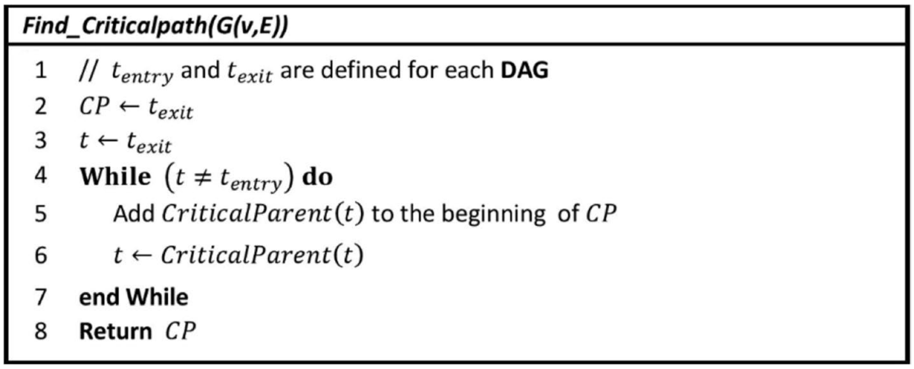
The workflow division algorithm follows different steps to divide the initial workflow into sub-workflows. Similar to DQWS [5], the proposed algorithm (Fig. 3) first determines the workflow's critical path. This is achieved by the *Find_CriticalPath* function shown in Fig. 4. In determining the critical path, it is important to identify the critical parent of each task. The critical parent of $t_i$ is one of its parents, such as $t_p$, which maximizes the expression, $EFT(t_p) + CM_{p,\,i}$. $CM_{p,\,i}$ is the communication time between $t_p$ and $t_i$.

After determining a critical path, a set of possible resources is defined to schedule it by the *Find_PossibleResources* function. This function searches among a variety of resource types offered by the service provider that are capable of scheduling the critical path tasks before the user-defined deadline and creates a set of these resource types as a possible resource set. The cheapest resource type from this set is then selected and the critical path tasks are pre-assigned on an instance of the selected resource. Because the critical



```
Workflow Division Algorithm
// let G=(V, E) be a DAG which represents a workflow with |V| tasks and |E| edges
Input $G$
$S_{nonlinear\_graph} = \{G\}$ , $S_{linear\_graph} = \{\ \}$
for all $(nonlinear\_graph_i \in S_{nonlinear\_graph})$ do
    $CP \leftarrow Find\_CriticalPath(nonlinear\_graph_i)$
    $S_{Resource} \leftarrow Find\_PossibleResources(CP)$
    Do
        If $S_{Resource} = \emptyset$ Return scheduling is not successful
        $R \leftarrow Resource_j$ // $Resource_j \in S_{Resource}$
        $S_{Resource} \leftarrow S_{Resource} - \{Resource_j\}$
        $PreSchedule(R, CP)$
        $check \leftarrow Check\_Subpaths(CP)$
    While $check = false$
    $Remove\_CP(nonlinear\_graph_i)$
    $S_{nonlinear-graph} \leftarrow S_{nonlinear-graph} \cup$ {all new nonlinear graph created after removing CP}
    $S_{linear\_graph} \leftarrow S_{linear\_graph} \cup$ {all new linear graph created after removing CP}
end for
end
```

**Fig. 3** Workflow division algorithm



```
Find_Criticalpath(G(v,E))
// $t_{entry}$ and $t_{exit}$ are defined for each DAG
$CP \leftarrow t_{exit}$
$t \leftarrow t_{exit}$
While $(t \neq t_{entry})$ do
    Add $CriticalParent(t)$ to the beginning of $CP$
    $t \leftarrow CriticalParent(t)$
end While
Return $CP$
```

**Fig. 4** Find_Criticalpath function

path is eliminated from the workflow in the next steps, it is necessary to finalize the scheduling of the critical path task. The *Check_Subpaths* function is responsible for finalizing critical path scheduling. To accomplish this, the *Check_Subpaths* function checks all sub-paths leading to the critical path and examines the possibility that each sub-path can be scheduled by at least one type of resource. The *Find_Subpaths* function determines the sub-paths leading to each critical task, as shown in Fig. 5.

The successful output of the *Check_Subpaths* function indicates that the critical path scheduling on the selected resource is finalized. Fig. 6 presents the details of this function. After finalization of the critical path schedule, the tasks in this path are removed from the

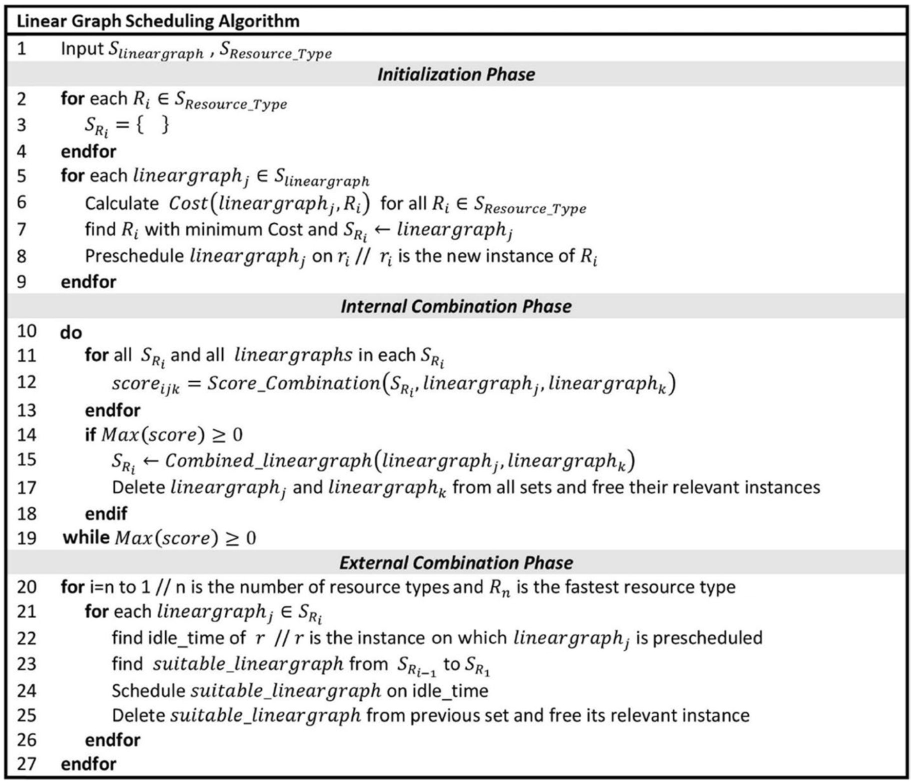
```
Find_Subpaths($t_{CP}$)
sp ← null
Add all unscheduled Parent of $t_{CP}$ to $ParentSet$
for all ($t_i \in ParentSet$) do
    Add $t_i$ to the beginning of $sp$
    while ( ($t_i$) has unscheduled Parent) do
        Add $CriticalParent(t_i)$ to the beginning of $sp$
        $t_i \leftarrow CriticalParent(t_i)$
    end While
    $subpath \leftarrow sp$
    $sp \leftarrow null$
end for
Return $subpath$
```

**Fig. 5** Find_Subpaths function.

```
Check_Subpaths(CP)
check=true
for all ($t_i \in CP$ and $t_i \neq$ first task of $CP$) do
    $S_{subpath} \leftarrow Find\_Subpaths(t_i)$
    for all ($sp_j \in S_{subpath}$) do
        Calculate $ST_{subpath}$, $FT_{subpath}$, runtime$_{subpath}$
        $S_{Resources} \leftarrow Find\_PossibleResources()$
        if $S_{Resources}$ is empty then
            Add the last task of $sp_j$ to CP before $t_i$
            Update ST and FT of all Critical tasks
            if FT of last Critical task > CP's Deadline then
                $check = false$
                return check
            end if
            $i \leftarrow i - 1$
            Exit from loop
        else
            Preassigned all task of $sp_j$ to the cheapest Resource of $S_{Resources}$
            Call Check_Subpaths($sp_j$)
        end if
    end for
end for
return check
```

**Fig. 6** Check_Subpaths function

workflow and the workflow is divided into one or more connected graphs. Depending on their structure, each of the resulting graphs is added to one of the linear graph or nonlinear graph sets. This operation is performed for all graphs in the nonlinear graph set.



```
Linear Graph Scheduling Algorithm
1  Input $S_{lineargraph}$ , $S_{Resource\_Type}$
                    Initialization Phase
2  for each $R_i \in S_{Resource\_Type}$
3      $S_{R_i} = \{\ \}$
4  endfor
5  for each $lineargraph_j \in S_{lineargraph}$
6      Calculate $Cost(lineargraph_j, R_i)$ for all $R_i \in S_{Resource\_Type}$
7      find $R_i$ with minimum Cost and $S_{R_i} \leftarrow lineargraph_j$
8      Preschedule $lineargraph_j$ on $r_i$ // $r_i$ is the new instance of $R_i$
9  endfor
                    Internal Combination Phase
10 do
11     for all $S_{R_i}$ and all $lineargraphs$ in each $S_{R_i}$
12         $score_{ijk} = Score\_Combination(S_{R_i}, lineargraph_j, lineargraph_k)$
13     endfor
14     if $Max(score) \geq 0$
15         $S_{R_i} \leftarrow Combined\_lineargraph(lineargraph_j, lineargraph_k)$
17         Delete $lineargraph_j$ and $lineargraph_k$ from all sets and free their relevant instances
18     endif
19 while $Max(score) \geq 0$
                    External Combination Phase
20 for i=n to 1 // n is the number of resource types and $R_n$ is the fastest resource type
21     for each $lineargraph_j \in S_{R_i}$
22         find idle_time of $r$ // $r$ is the instance on which $lineargraph_j$ is prescheduled
23         find $suitable\_lineargraph$ from $S_{R_{i-1}}$ to $S_{R_1}$
24         Schedule $suitable\_lineargraph$ on idle_time
25         Delete $suitable\_lineargraph$ from previous set and free its relevant instance
26     endfor
27 endfor
```

**Fig. 7** Linear graph scheduling algorithm

### Linear graph scheduling algorithm

Since the linear graphs obtained from different stages of the workflow division are independent of each other, there is no requirement to observe a specific order in their execution. However, a specific arrangement or combination in their scheduling can be effective in choosing the resource type, which, in turn, will affect the workflow execution cost. In contrast to the current authors' previous study (DQWS) [5], which employs a greedy method to schedule linear graphs, the present paper utilizes a scoring function for its linear graph scheduling. As shown in Fig. 7, the proposed algorithm consists of three different phases: the initialization phase, the internal combination phase, and the external combination phase. The following presents the details of these different algorithm phases.

#### *Initialization phase*

The objective of the algorithm initialization phase (Lines 2–9) is to determine the candidate resources for each linear graph and distribute the linear graphs in the sets associated with their candidate resources. A candidate resource is a resource type whose linear graph execution cost is the lowest among the other resource types. Each linear graph may have more than one candidate resource. At the beginning of this phase, a dependent set , $S_{R_i}$, is defined for each resource type, $R_i$, offered by the service provider and, for each linear graph, candidate resources are determined. Then, each linear graph is pre-scheduled on a new instance of each candidate resource type and then added to the sets that belong to its candidate resources.

#### *Internal combination phase*

The objective of the proposed algorithm's second phase (Lines 10–19) is to reduce the workflow execution cost, as performed by pairwise combinations of linear graphs. During the process of combination, the preference of the algorithm is lowering the cost and, as a result, reducing the number of instances required for scheduling. In this phase, in each of the defined sets, $S_{R_i}$, all pairwise combinations of linear graphs are determined and the score of each combination is calculated by the *Score_Combination* function. As shown in (9), the combined score is calculated as the difference in execution costs when both

```
Score_Combination(linear_graph_1, linear_graph_2)
// $D1$ is the deadline of $linear\_graph_1$, $D2$ is the deadline of $linear\_graph_2$
// $cost_{\,linear\_graph_1}$ is the cost of $linear\_graph_1$ before combination
// $cost_{\,linear\_graph_2}$ is the cost of $linear\_graph_2$ before combination
if $\left(\sum_{t_i \in linear\_gaph_1} W(t_i) + \sum_{t_j \in linear\_gaph_2} W(t_j)\right) > D_1\ and\ D_2$
    Combination of two graphs is impossible
    Combination Cost =inf
else
    if $(EST(first\ task\ of\ linear\_graph_1) > D_2)\ or\ (EST(first\ task\ of\ linear\_graph_2) > D_1)$
        Combine two graphs without changing the start time and finish time of tasks
    else
        Combine two graphs by modified merge sort
    end if
    Calculate Combination Cost by Equation (2)
end if
Score=cost $_{linear\_graph_1}$ +cost $_{linear\_graph_2}$- Combination Cost
return Score
```

**Fig. 8** Score combination function

linear graphs are scheduled on one sample and when each is scheduled on a separate sample. graphs is explored. If this is not possible, the combination cost is considered infinite.

$$\text{Score} = cost_{linear\_graph_1} + cost_{linear\_graph_2} - Combination\ Cost \tag{9}$$

Due to the large number of linear graphs in each set, determining all pairwise combinations of the linear graphs as well as calculating the score of each combination is quite time consuming. For this reason, a parallel algorithm performs this function. Subsection 4–2–2-1 provides the details of the *Score_Combination* function. After the score of all combinations is calculated, the combination with the highest score and its set are determined and the combined graph is added to the selected set. Then the initial linear graphs making up the combined graph are removed from all sets. Since the sets are not disjoint, the initial linear graphs must be removed from all the sets. Determining the pairwise combinations of linear graphs and selecting the best combination is repeated until the score of all combinations in all sets is negative.

*Details of the scoring function* The score of each binary combination is calculated by the *Score_Combination* function. Figure 8 presents the pseudo-code of this function. At first, the possibility of combining two linear graphs is explored. If this is not possible, the combination cost is considered infinite.

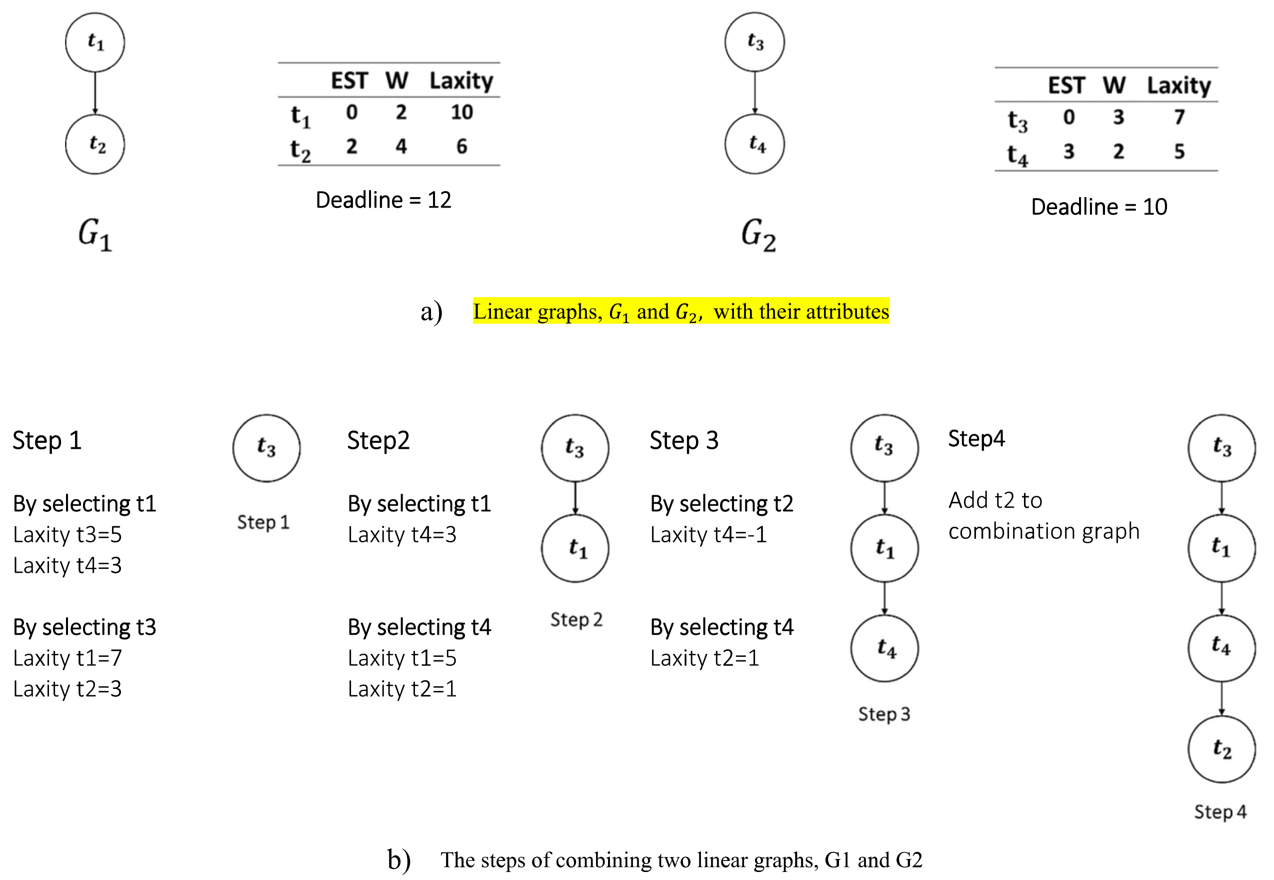
> **Definition:** $G_1$ and $G_2$ are linear graphs and $D_1$ and $D_2$ are their deadlines, respectively. These linear graphs are *uncombinable* if the total execution time of their tasks is greater than both $D_1$ and $D_2$.

To determine the execution cost of the combinable graphs, two linear graphs are first pre-scheduled on one instance. Pre-scheduling is performed in two different ways: pre-scheduling for graphs with a time overlap and pre-scheduling for graphs without a time overlap.

> **Definition:** Linear graphs, G1 and G2, are not *time-overlapped graphs* if:

$$\left\{\begin{array}{c} EST(G_2) \geq D_1 \\ or \\ EST(G_1) \geq D_2 \end{array}\right. \tag{10}$$

In this case, by pre-scheduling the tasks of both graphs on one instance, the order, start time, and finish time of the tasks does not change.



| | EST | W | Laxity |
|---|---|---|---|
| $t_1$ | 0 | 2 | 10 |
| $t_2$ | 2 | 4 | 6 |

Deadline = 12

| | EST | W | Laxity |
|---|---|---|---|
| $t_3$ | 0 | 3 | 7 |
| $t_4$ | 3 | 2 | 5 |

Deadline = 10

**Fig. 9** Illustrative example for combining two linear graphs

To pre-schedule linear graphs with a time overlap, the present study provides a new merge list algorithm. The proposed merge list algorithm receives two lists as input. Each list includes the tasks of one linear graph that are arranged from entry task to exit task. The output of this algorithm will also be a list that specifies the final sequence of the tasks in the combined graph. The current work's contribution to the merge list algorithm is the header selection method. To select a header, the laxity of all tasks in both graphs is first determined and the following conditions are considered:

a. Selecting each header creates negative laxity for one or more tasks of the other list.
b. Selecting one of the headers creates negative laxity for one or more tasks of the other list.
c. Neither of the choices leads to negative laxity.

In case (a), it is not possible to combine two linear graphs; hence, the algorithm terminates. In case (b), a header is selected that does not create a negative laxity. In case (c), a header is chosen with an earlier start time in its initial linear graph and, if the start time of both headers is the same, then a header with lower laxity is selected. Fig. 9 shows an illustrative example for combining two linear graphs, $G_1$ and $G_2$, with the proposed modified merge list algorithm.

After examining the possibility of combining linear graphs, the execution cost of the combined graphs is calculated by (7).

### *External combination phase*

In this phase (Lines 20–27), the proposed algorithm aims to transfer the scheduled tasks of the less powerful instances to the idle times of the more powerful ones. This transfer can reduce the number of used instances and consequently the overall execution cost. For this purpose, in each stage, the idle-time periods of the most powerful utilized instance are determined. For each idle time, the present study looks for an appropriate linear graph that can be scheduled on the idle time before its

**Table 1** VM types based on Amazon EC2

| Type | ECU | Memory (GB) | Cost($)(per hour) |
|---|---|---|---|
| m3.medium | 3 | 3.75 | 0.067 |
| m4.large | 6.5 | 8 | 0.126 |
| m3.xlarge | 13 | 15 | 0.266 |
| m4.2xlarge | 26 | 32 | 0.504 |
| m4.4xlarge | 53.5 | 64 | 1.008 |

deadline. After the desired linear graph is determined, its tasks are transferred and the original instance is removed from the required instances.

## Evaluation

This section first describes the current work's experimental setting and evaluation criteria, and then compares the proposed method with state-of-the-art approaches that are similar in terms of objectives. The methods used for comparison are IC-PCP [18], BDHEFT [15], PDC [12], and the present authors' previous study, DQWS [5]. All of these algorithms are reviewed in the Related Work section.

### Experimental setting and evaluation criteria

To evaluate the performance of the proposed algorithm, different simulation scenarios are run. Simulation is a well-accepted approach for evaluating workflow scheduling algorithms. With simulation, it is possible to test the performance of the algorithms under a controlled setting [27] . In the present study's simulation, which is performed by MATLAB, the service provider offers five different VM types with different costs and processing powers. The characteristics of the VMs are based on the US-east Amazon region and are presented in Table 1. Unlimited instances of each VM type are assumed. VMs are in a single data center and the average bandwidth between instances is fixed at 20 MBps [28].

The current study conducts its experiments using four different scientific workflows: CyberShake, Epigenomics, LIGO, and Motif. These workflows are diverse in terms of structure, computational characteristics, and communication data. In addition, the workflows are employed in different scientific areas, such as earthquake science, biology, gravitational physics, and genetics. Full descriptions of these workflows are provided in [29, 30]. Fig. 10 depicts the structure of each workflow in a relatively small size. These workflows are provided by the Pegasus workflow generator[4] as a DAX (Directed Acyclic Graph in XML) format and, for each workflow, the details, including the DAG, the sizes of data transfer, and the tasks execution time, are published. These workflows have been widely used for evaluating the performance of scheduling algorithms, and thus they are included in the current study's experiments. In order to determine the execution time of each workflow task by each VM type, we assume the published execution time for each task is calculated for the VM with ECU = 1. With this assumption, the execution time of each task on the other VM types is calculated by dividing its published execution time by the ECU value of the VM which is shown in Table 1. As shown in Table 2, the present work chooses four different sizes for each workflow type and generates 50 random samples from each type and size of workflow.

To assess the performance of the compared algorithms, it is necessary to determine the acceptable values for each workflow's deadline. The concept of the fastest schedule is utilized to determine deadlines proportionate to each workflow structure. The fastest schedule is the scheduling of each workflow task on a distinct instance of the fastest VM type without considering the communication time between tasks [18]. The makespan obtained by the fastest schedule for each workflow, $makespan_{LB}$, is the lower bound of the overall workflow execution time. Using $makespan_{LB}$, the present study calculates a variation for a deadline, from tight to relaxed, as follows:

$$dealline = \alpha \times makespan_{LB} 2 \leq \alpha \leq 10 \tag{11}$$

α starts from 2 and is increased by 2 up to a value of 10. In this way, the impact of different deadlines (from tight to relaxed) is evaluated on the performance of each algorithm.

To evaluate the performance of the algorithm, the present work analyzes the following metrics in its experiments: normalized cost, success rate, and instance reduction. Each metric is explained as follows:

> **Normalized cost**: The overall workflow execution cost obtained by each of the algorithms is a suitable criterion for evaluating the compared algo-

**Table 2** Different sizes of workflows according to the number of tasks

| Scientific Workflow | Small | Medium | Large | Extra Large |
|---|---|---|---|---|
| CyberShake | 30 | 50 | 100 | 500 |
| Epigenomics | 24 | 46 | 100 | 500 |
| LIGO | 30 | 50 | 100 | 500 |
| Motif | 30 | 50 | 100 | – |

[4] https://confluence.pegasus.isi.edu/display/pegasus/

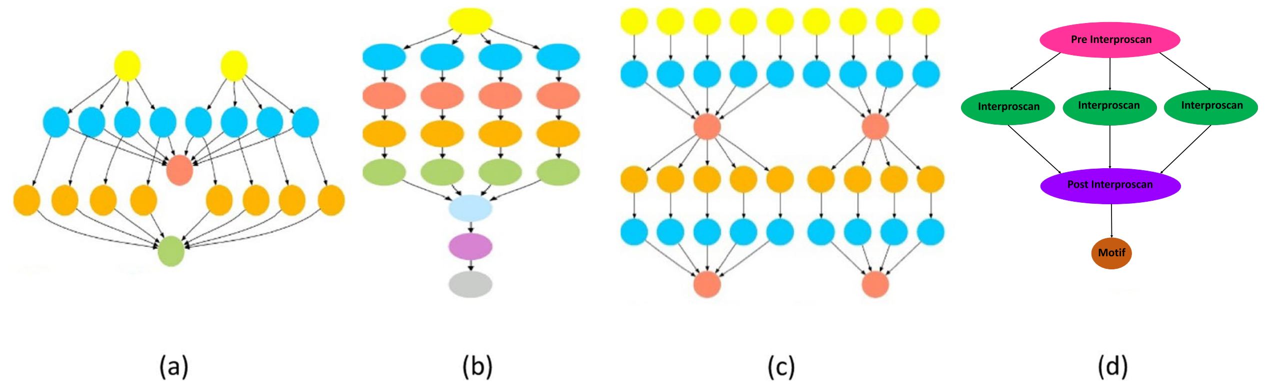


**Fig. 10** Structure of the examined workflow [54]. **a** CyberShake. **b** Epigenomics. **c** LIGO. **d** Motif

rithms. Because of the difference in the structure and characteristics of the benchmark workflows, the current paper employs a normalized cost metric for comparison. The normalized cost of executing each workflow with any of the benchmark algorithms is obtained by dividing its overall execution cost, *cost*(*wf*), by its scheduling cost based on the cheapest scheduling strategy($cost_{lowest}(wf)$) [18], as follows:

$$Normalized\ Cost(wf) = \frac{cost(wf)}{cost_{lowest}(wf)} \tag{12}$$

**Success rate:** To evaluate the capability of each algorithm to meet the deadline constraints, the present study defines the metric of success rate as the ratio between the number of successful schedules and the total number of schedules.

**Instance reduction:** To compare the performance of the proposed algorithm against that of the current authors' previous research (DQWS), an instance reduction parameter is introduced. This parameter shows the reduction factor in the percentage of the number of instances needed for scheduling each workflow type by EDQWS in comparison to DQWS. Since the present study's focus in the linear graph scheduling phase is the combination of linear graphs, reducing the number of instances, in addition to minimizing the cost, can serve as a metric for comparing the two methods. A reduction in the number of instances required to schedule workflows can lower the probability of instance failure. This is also helpful for cloud providers which have a limitation in the number of requested instances.

## Experimental result

The performance of the proposed algorithm is evaluated in three separate subsections. In the first subsection, the execution cost of each workflow type is examined by all compared algorithms. In the second subsection, the success rate of all compared algorithms is evaluated. In the third subsection, the two methods, DQWS and EDQWS, are compared in more detail. For this purpose, a comparison is made between the average cost reduction in EDQWS and in DQWS. Additionally, the number of instances required for each algorithm to schedule each workflow type is studied.

### *Normalized cost analysis*

In this subsection, the execution cost of each workflow is examined by all the compared algorithms. In these experiments, 50 random samples are generated from each type and size of the workflows, and each random sample is scheduled by all algorithms for five different deadline factors. Given that the results of running each algorithm on different workflow sizes are relatively similar, only the results of implementing the largest size of each workflow type are reported. Fig. 11 presents the normalized cost of all of the compared algorithms when executing the four workflow types (CyberShake, Epigenomics, LIGO, and Motif). The normalized cost values are the mean value of the 50 random samples generated for each workflow type. Besides the mean values, the standard deviation is also provided in Fig. 11. The reporting of 'Fail' in each graph indicates that the specified algorithm was unable to schedule the workflow within the given deadline factor.

Figure 11.a presents the results of the CyberShake workflow execution by the five compared algorithms. As observed in Fig. 11.a, none of the compared algorithms are able to schedule this type of workflow within the deadline factor of 2. The lowest execution cost for this type of workflow is obtained by the EDQWS algorithm.

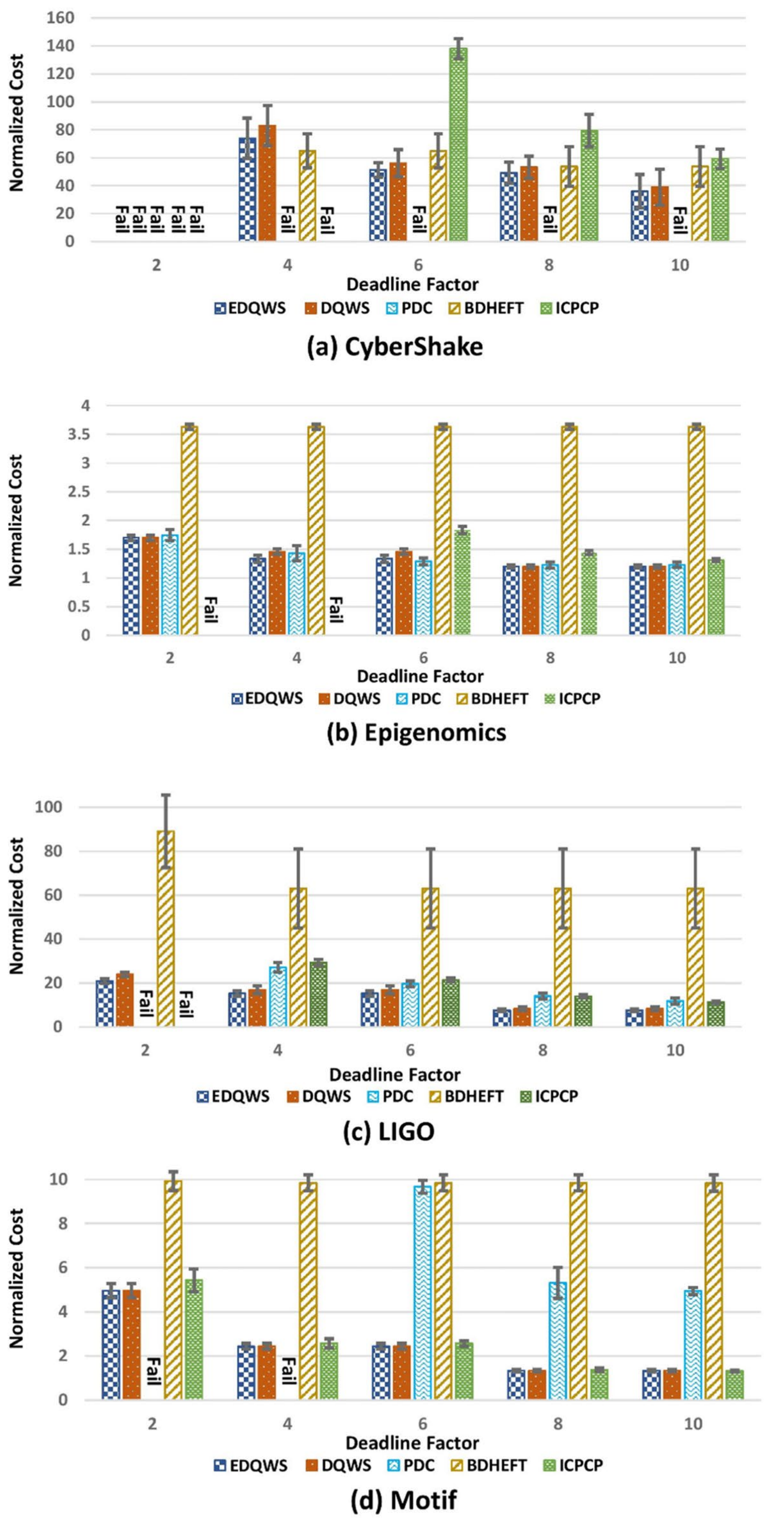


**Fig. 11** Normalized Cost vs. Deadline Factor for four different workflow sets (PDC was unable to schedule the CyberShake workflow). Note: the error bar indicates the standard deviation

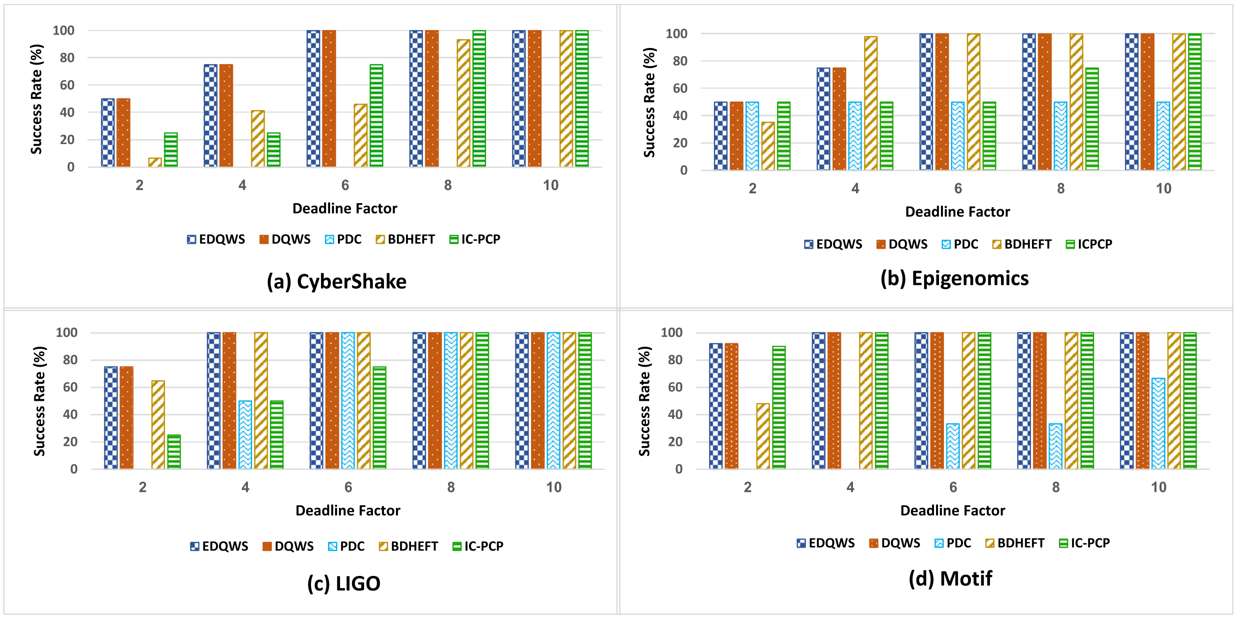


**Fig. 12** Success rates of the compared algorithms on the four workflow sets

In this type of workflow, DQWS and EDQWS succeed in creating multiple linear graphs with a small number of tasks and short execution times. As a result, utilizing the idle time of instances for scheduling linear graphs can reduce the execution cost of this workflow type.

Although the two algorithms utilize the same linear graphs, a comparison of their normalized cost reduction indicates that EDQWS' consideration of different linear graph combinations in instance type selection leads to more appropriate solutions than DQWS' usage of the greedy method.

As seen in Fig. 11.a, the PDC algorithm failed to schedule any random samples of the CyberShake workflow within the specified deadlines. The failure of PDC in scheduling this type of workflow is due to the way in which it sets the initial deadline. PDC determines the initial deadline for each workflow by considering the communication time between tasks. If the user-defined deadline is sooner than the initial deadline, the workflow will not be scheduled. As mentioned earlier, for determining the deadline, the proposed algorithm employs the concept of the fastest schedule which eliminates the communication time between tasks during the deadline calculation. Because the execution time of CyberShake tasks are short and the communication time between them is long, the deadlines calculated by the fastest schedule are shorter than the PDC initial deadline. Therefore, the results of the PDC algorithm are not included in Fig. 11.a.

In Epigenomics, the possibility that DQWS and EDQWS will utilize the resource idle time is less than that of the other workflows because of the high execution time of tasks. As seen in Fig. 11.b, the normalized cost obtained by DQWS, EDQWS, and PDC is almost the same, while BDHEFT reports a higher execution cost than the other algorithms. For the deadline factors that IC-PCP is successful in meeting, its normalized execution cost nears that of DQWS, EDQWS, and PDC.

Figure 11.c shows the results of LIGO scheduling by all of the compared algorithms. In this type of workflow, the normalized cost obtained by EDQWS is the lowest among the other algorithms. Due to the existence of several parallel tasks in the structure of this workflow, the creation of multiple linear graphs has increased the possibility of utilizing instance idle times. Furthermore, the usage of the scoring method and examination of linear graph combinations may be the reason why the normalized cost reduction in EDQWS is greater than that of the greedy method.

Figure 11.d provides the results obtained from the present study's simulations for the Motif workflow. As seen, IC-PCP, DQWS, and EDQWS outperform the BDHEFT and PDC algorithms. Based on the results obtained from IC-PCP, DQWS, and EDQWS, it is observed that, in tight deadlines, the DQWS and EDQWS algorithms perform better while, in relaxed deadlines, ICPCP's performance is superior. For motif scheduling, EDQWS performs much like DQWS and the normalized cost of these two algorithms is the same. A possible reason for this similarity may be that the motif structure creates single-task linear graphs during the division phase. Subsection 5–2-3 shall compare these two methods in more detail.

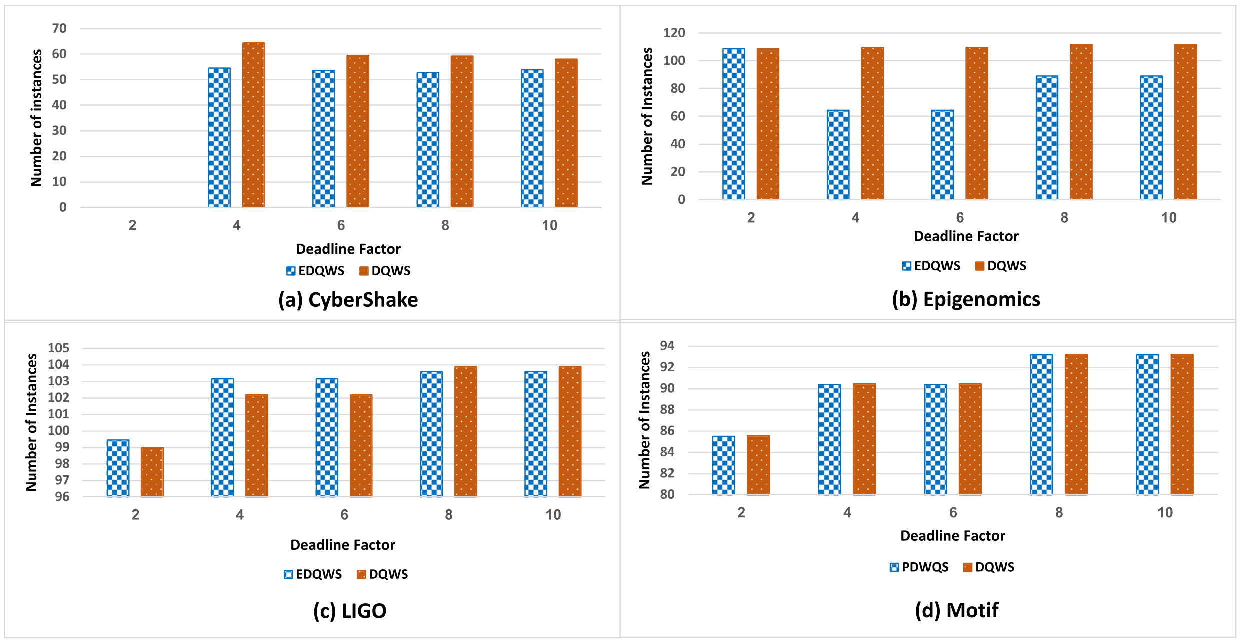


**Fig. 13** Number of Instances vs. Deadline Factor for four different workflow sets

### *Success rate analysis*

By considering different deadline factors, this section examines the success rate of the compared algorithms for each workflow type. All random samples created from any type of workflow are included in the success rate calculation. Fig. 12 provides the success rates of the five algorithms (EDQWS, DQWS, PDC, BDHEFT, and IC-PCP). As the results show, the success rates of EDQWS and DQWS are exactly the same for all deadline factors and the different workflows. Given that these two algorithms employ the same method to create linear graphs, it can be concluded that the method of scheduling linear graphs does not impact the success or failure of the schedule but only affects the workflow execution cost.

As shown in Fig. 12, for all workflow types, the proposed method has a higher success rate than that of the PDC, IC-PCP, and BDHEFT algorithms. PDC's low success rate may be explained by the fact that this algorithm does not accept most of the deadline factors and so is unable to schedule the workflow.

### *The comparison of DQWS and EDQWS*

This section compares the current authors' newly introduced EDQWS algorithm and their previous algorithm, DQWS. As mentioned earlier, the steps for creating linear graphs in these two algorithms are similar, but their methods differ in the linear graph scheduling phase. For scheduling linear graphs, DQWS utilizes a greedy method while the present paper's EDQWS algorithm employs a scoring method. To evaluate the performance of these two algorithms, two comparisons are made. First, the number of instances required by EDQWS and DQWS to schedule each workflow type is compared. Second, the difference between these two algorithms' normalized cost and the number of instances are examined.

Figure 13 shows the number of instances required by DQWS and EDQWS to schedule the CyberShake, Epigenomics, LIGO, and Motif workflows. It should be noted that the number of instance values is the mean value of 50 random samples generated for each workflow type. As seen in Fig. 13, in comparison to DQWS, the number of instances required for scheduling CyberShake and Epigenomics is greatly reduced by the EDQWS algorithm. However, this reduction is not observed in the LIGO and Motif workflows.

Table 3 summarizes the results obtained from the above experiments. This table presents three different parameters for each algorithm: the average percentage of cost reduction in EDQWS compared to DQWS, the average percentage of the reduction in the number of instances in EDQWS versus DQWS, and the percentage of experiments in which EDQWS succeeds in reducing the number of instances in comparison to DQWS.

As seen in Table 3, the average normalized cost for the three types of workflow (CyberShake, Epigenomics, and LIGO) is reduced by the proposed algorithm when compared to the DQWS algorithm. Also, the current

**Table 3** Comparison of EDQWS and DQWS in the number of instances and normalized cost

| Workflow | Average reduction in the normalized cost of EDQWS vs DQWS | Average reduction in the number of instances of EDQWS vs DQWS | Experiments that decrease the number of instances of EDQWS vs DQWS |
|---|---|---|---|
| CyberShake | 8.6% | 10.98% | 68.92% |
| Epigenomics | 3.4% | 24.58% | 81.6% |
| LIGO | 10.5% | −0.36% | 51.6% |
| Motif | 0% | 0.05% | 8.4% |

work's algorithm succeeds in reducing the average number of instances required to schedule CyberShake and Epigenomics. In LIGO, although the average number of instances in the proposed algorithm increases slightly, the number of instances in more than half of the experiments decreases in comparison to DQWS. In Motif, despite the same normalized cost in both algorithms, the number of instances decreases in a small percentage of the proposed method's experiments.

## Conclusion

The present paper proposes a static scheduling algorithm called *EDQWS,* which is an extension of a previous study by the current authors [5]. *EDQWS* is a two-phase workflow scheduler based on divide and conquer and aims to minimize the overall workflow execution cost by considering a user-defined deadline. In the first phase, similar to the present authors' previous research, the division of workflow into sub-workflows is achieved by determining and scheduling the critical path and removing it from the workflow. By eliminating the critical path, the workflow is divided into several sub-workflows, each of which undergoes this same division. The stop condition is to attain a sub-workflow with a chain structure called a linear graph. For scheduling linear graphs in the second phase, the current work proposes a new merging algorithm to combine the resulting linear graphs, reduce the number of used instances, and minimize the overall execution cost. Also introduced is a scoring function to select the most efficient instances for scheduling the linear graphs.

The experiments are conducted with four well-known workflows that determine whether EDQWS has an overall better performance than the state-of-the-art algorithms, IC-PCP, PDC, BDHEFT, and DQWS. In terms of the normalized cost parameter, EDQWS shows acceptable results when compared to the other methods. As for the success rate parameter, EDQWS and DQWS are completely the same in all deadline factors and for different workflows. Given that these two algorithms use the same method in creating linear graphs, the method of scheduling linear graphs has no effect on the success or failure of the schedule and only affects the workflow execution cost. However, the success rate of both algorithms is higher than that of the other methods, especially under tight deadlines. In comparing the performance of EDQWS with that of the present authors' previous research (DQWS), the results show that, in more than 50% of the examined workflow samples, the number of resource instances decreases in EDQWS in comparison to DQWS. Reducing the number of resource instances in addition to decreasing the probability of instance failure also leads to a reduction in the overall execution cost of the examined workflows. According to the above description, it can be concluded that the definition of the scoring function and relying on it to combine linear graphs and select virtual machine types has led to a more appropriate selection of VM types than the other baseline methods. The use of our new merge list algorithm as well as the policy for transferring the task from the less powerful instances to the idle times of the more powerful ones has also had a significant impact on improving the results. The new merge list algorithm combines the tasks of linear graphs with respect to the laxity of tasks and improves the scheduling of tasks by changing the mapping of pre-scheduled tasks on the instances. By transferring the pre-scheduled tasks from the less powerful instances to the idle times of the more powerful ones, the algorithm can use these idle times for which there is no need to pay extra, and remove less powerful instances from the list of required instances. For future work, the current authors intend to extend their merging algorithm for more than two linear graphs. Furthermore, a non-greedy algorithm shall be proposed for selecting instance idle-time to transfer scheduled tasks in the external combination phase.

**Abbreviations**
DAG: Directed Acyclic Graph; MIP: Mixed Integer Programing; VM: Virtual Machine; CP: Critical Path; DAX: Directed Acyclic Graph in XML.

**Acknowledgments**
Not applicable.

**Authors' contributions**
All authors read and approved the final manuscript.

**Funding**
This work has no funding.

**Availability of data and materials**
The data used during the current study are available from the corresponding author on reasonable request.

## Declarations

**Competing interests**
The authors declare that they have no competing interests.

**Author details**
[1]Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran. [2]Khorasan Electric Distribution Company (KEDC), Mashhad, Iran.

Received: 6 September 2021 Accepted: 21 March 2022
Published online: 26 May 2022

**References**
1. Guo W, Lin B, Chen G, Chen Y, Liang F (2018) Cost-driven scheduling for deadline-based workflow across multiple clouds. IEEE Trans Netw Serv Manag 15(4):1571–1585. https://doi.org/10.1109/TNSM.2018.2872066
2. Wu Q, Ishikawa F, Zhu Q, Xia Y, Wen J (2017) Deadline-constrained cost optimization approaches for workflow scheduling in clouds. IEEE Trans Parallel Distributed Syst 28(12):3401–3412. https://doi.org/10.1109/TPDS.2017.2735400
3. Rodriguez MA, Buyya R (2017) A taxonomy and survey on scheduling algorithms for scientific workflows in IaaS cloud computing environments. Concurrency Comput 29(8). https://doi.org/10.1002/cpe.4041
4. Faragardi HR, Saleh Sedghpour MR, Fazliahmadi S, Fahringer T, Rasouli N (2020) GRP-HEFT: a budget-constrained resource provisioning scheme for workflow scheduling in IaaS clouds. IEEE Trans Parallel Distributed Syst 31(6):1239–1254. https://doi.org/10.1109/TPDS.2019.2961098
5. Khojasteh Toussi G, Naghibzadeh M (2021) A divide and conquer approach to deadline constrained cost-optimization workflow scheduling for the cloud. Clust Comput. https://doi.org/10.1007/s10586-020-03223-x
6. Singh V, Gupta I, Jana PK (2020) An energy efficient algorithm for workflow scheduling in IaaS cloud. J Grid Comput 18(3):357–376. https://doi.org/10.1007/s10723-019-09490-2
7. Garg N, Singh D, Goraya MS (2021) Energy and resource efficient workflow scheduling in a virtualized cloud environment. Clust Comput 24(2):767–797. https://doi.org/10.1007/s10586-020-03149-4
8. Jiang J, Lin Y, Xie G, Fu L, Yang J (2017) Time and energy optimization algorithms for the static scheduling of multiple workflows in heterogeneous computing system. J Grid Comput 15(4):435–456. https://doi.org/10.1007/s10723-017-9391-5
9. Sreenu K, Sreelatha M (2019) W-scheduler: whale optimization for task scheduling in cloud computing. Clust Comput 22:1087–1098. https://doi.org/10.1007/s10586-017-1055-5
10. Wang S, Li K, Mei J, Xiao G, Li K (2017) A reliability-aware task scheduling algorithm based on replication on heterogeneous computing systems. J Grid Comput 15(1):23–39. https://doi.org/10.1007/s10723-016-9386-7
11. Kalyan Chakravarthi K, Shyamala L, Vaidehi V (2020) Budget aware scheduling algorithm for workflow applications in IaaS clouds. Clust Comput 23(4):3405–3419. https://doi.org/10.1007/s10586-020-03095-1
12. Arabnejad V, Bubendorfer K, Ng B (2017) Scheduling deadline constrained scientific workflows on dynamically provisioned cloud resources. Futur Gener Comput Syst 75:348–364. https://doi.org/10.1016/j.future.2017.01.002
13. Rizvi N, Ramesh D (2020) Fair budget constrained workflow scheduling approach for heterogeneous clouds. Clust Comput 23(4):3185–3201. https://doi.org/10.1007/s10586-020-03079-1
14. Cao, S., Deng, K., Ren, K., Li, X., Nie, T., and Song, J.: 'A deadline-constrained scheduling algorithm for scientific workflows in clouds', in Editor (Ed.)^(Eds.): 'Book A deadline-constrained scheduling algorithm for scientific workflows in clouds' (Institute of Electrical and Electronics Engineers Inc., 2019, edn.), pp. 98–105
15. Verma A, Kaushal S (2015) Cost-time efficient scheduling plan for executing workflows in the cloud. J Grid Comput 13(4):495–506. https://doi.org/10.1007/s10723-015-9344-9
16. Ullman JD (1975) NP-complete scheduling problems. J Comput Syst Sci 10(3):384–393. https://doi.org/10.1016/S0022-0000(75)80008-0
17. Malawski M, Figiela K, Bubak M, Deelman E, Nabrzyski J (2015) Scheduling multilevel deadline-constrained scientific workflows on clouds based on cost optimization. Sci Program 2015. https://doi.org/10.1155/2015/680271
18. Abrishami S, Naghibzadeh M, Epema DHJ (2013) Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. Futur Gener Comput Syst 29(1):158–169. https://doi.org/10.1016/j.future.2012.05.004
19. Calheiros RN, Buyya R (2014) Meeting deadlines of scientific workflows in public clouds with tasks replication. IEEE Trans Parallel Distributed Syst 25(7):1787–1796. https://doi.org/10.1109/TPDS.2013.238
20. Arabnejad, V., Bubendorfer, K., Ng, B., and Chard, K.: 'A Deadline Constrained Critical Path Heuristic for Cost-Effectively Scheduling Workflows C3 - Proceedings - 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015', in Editor (Ed.)^(Eds.): 'Book A Deadline Constrained Critical Path Heuristic for Cost-Effectively Scheduling Workflows C3 - Proceedings - 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015' (Institute of Electrical and Electronics Engineers Inc., 2015, edn.), pp. 242–250
21. Rodriguez MA, Buyya R (2014) Deadline based resource provisioningand scheduling algorithm for scientific workflows on clouds. IEEE Trans Cloud Comput 2(2):222–235
22. Arabnejad H, Barbosa JG (2014) A budget constrained scheduling algorithm for workflow applications. J Grid Comput 12(4):665–679. https://doi.org/10.1007/s10723-014-9294-7
23. Topcuoglu H, Hariri S, Wu MY (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distributed Syst 13(3):260–274. https://doi.org/10.1109/71.993206
24. Wu F, Wu Q, Tan Y, Li R, Wang W (2016) PCP-B2: partial critical path budget balanced scheduling algorithms for scientific workflow applications. Futur Gener Comput Syst 60:22–34. https://doi.org/10.1016/j.future.2016.01.004
25. Durillo JJ, Prodan R (2014) Multi-objective workflow scheduling in amazon EC2. Clust Comput 17(2):169–189. https://doi.org/10.1007/s10586-013-0325-0
26. Wu, Z., Ni, Z., Gu, L., and Liu, X. (2010). 'A revised discrete particle swarm optimization for cloud workflow scheduling C3 - Proceedings - 2010 International Conference on Computational Intelligence and Security, CIS 2010', in Editor (Ed.)^(Eds.): 'Book A revised discrete particle swarm optimization for cloud workflow scheduling C3 - Proceedings - 2010 International Conference on Computational Intelligence and Security, CIS 2010', pp. 184–188
27. Arabnejad V, Bubendorfer K, Ng B (2019) Budget and deadline aware e-science workflow scheduling in clouds. IEEE Trans Parallel Distributed Syst 30(1):29–44. https://doi.org/10.1109/TPDS.2018.2849396
28. Palankar, M.R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. (2008). 'Amazon S3 for science grids: a viable solution?', in Editor (Ed)^(Eds): 'Book Amazon S3 for science grids: a viable solution?', pp. 55–64
29. Juve G, Chervenak A, Deelman E, Bharathi S, Mehta G, Vahi K (2013) Characterizing and profiling scientific workflows. Futur Gener Comput Syst 29(3):682–692. https://doi.org/10.1016/j.future.2012.08.015
30. Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.H., and Vahi, K. (2008). 'Characterization of scientific workflows C3–2008 3rd Workshop on Workflows in Support of Large-Scale Science, WORKS 2008', in Editor (Ed.)^(Eds.): 'Book Characterization of scientific workflows C3–2008 3rd Workshop on Workflows in Support of Large-Scale Science, WORKS 2008', pp