Future Generation Computer Systems 29 (2013) 158–169

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds

Saeid Abrishami [a,*], Mahmoud Naghibzadeh [a], Dick H.J. Epema [b]

[a] Department of Computer Engineering, Engineering Faculty, Ferdowsi University of Mashhad, Azadi Square, Mashhad, Iran

[b] Parallel and Distributed Systems Group, Faculty EEMCS, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands

ARTICLE INFO

Article history:
Received 3 May 2011
Received in revised form
21 November 2011
Accepted 14 May 2012
Available online 23 May 2012

Keywords:
Cloud computing
IaaS Clouds
Grid computing
Workflow scheduling
QoS-based scheduling

ABSTRACT

The advent of Cloud computing as a new model of service provisioning in distributed systems encourages researchers to investigate its benefits and drawbacks on executing scientific applications such as workflows. One of the most challenging problems in Clouds is workflow scheduling, i.e., the problem of satisfying the QoS requirements of the user as well as minimizing the cost of workflow execution. We have previously designed and analyzed a two-phase scheduling algorithm for utility Grids, called Partial Critical Paths (PCP), which aims to minimize the cost of workflow execution while meeting a user-defined deadline. However, we believe Clouds are different from utility Grids in three ways: on-demand resource provisioning, homogeneous networks, and the pay-as-you-go pricing model. In this paper, we adapt the PCP algorithm for the Cloud environment and propose two workflow scheduling algorithms: a one-phase algorithm which is called IaaS Cloud Partial Critical Paths (IC-PCP), and a two-phase algorithm which is called IaaS Cloud Partial Critical Paths with Deadline Distribution (IC-PCPD2). Both algorithms have a polynomial time complexity which make them suitable options for scheduling large workflows. The simulation results show that both algorithms have a promising performance, with IC-PCP performing better than IC-PCPD2 in most cases.

1. Introduction

Cloud computing [1] is the latest emerging trend in distributed computing that delivers hardware infrastructure and software applications as services. The users can consume these services based on a *Service Level Agreement (SLA)* which defines their required Quality of Service (QoS) parameters, on a pay-as-you-go basis. Although there are many papers that address the problem of scheduling in traditional distributed systems like Grids, there are only a few works on this problem in Clouds. The multiobjective nature of the scheduling problem in Clouds makes it difficult to solve, especially in the case of complex jobs like workflows. Furthermore, the pricing model of the most current commercial Clouds, which charges users based on the number of time intervals that they have used, makes the problem more complicated. In this paper we propose two workflow scheduling algorithms for the Cloud environment by adapting our previous algorithm for utility Grids, called Partial Critical Paths (PCP), and we evaluate their performance on some well-known scientific workflows.

Workflows constitute a common model for describing a wide range of scientific applications in distributed systems [2]. Usually, a workflow is described by a Directed Acyclic Graph (DAG) in which each computational task is represented by a node, and each data or control dependency between tasks is represented by a directed edge between the corresponding nodes. Due to the importance of workflow applications, many Grid projects such as Pegasus [3], ASKALON [4], and GrADS [5] have designed workflow management systems to define, manage, and execute workflows on the Grid.

Recently, some researchers consider the benefits of using Cloud computing for executing scientific workflows [6–8]. Currently, there exist several commercial Clouds, such as Amazon, which provide virtualized computational and storage hardware on top of which the users can deploy their own application and services. This new model of service provisioning in distributed systems, which is known as Infrastructure as a Service (IaaS) Clouds, has some potential benefits for executing scientific workflows [7,8]. First, users can dynamically obtain and release resources on demand, and they will be charged on a pay-as-you-go basis. This helps the workflow management system to increase or decrease its acquired resources according to the needs of the workflow and the user's deadline and budget. The second advantage of the Clouds is direct resource provisioning versus the best-effort method in providing resources in community Grids. This feature can significantly improve the performance of scheduling interdependent tasks of a workflow. The third advantage is the illusion of unlimited resources [7]. It means the user can ask for any resource whenever he needs it, and he will certainly (or with a very high possibility)

* Corresponding author.
E-mail addresses: s-abrishami@um.ac.ir (S. Abrishami), naghibzadeh@um.ac.ir (M. Naghibzadeh), d.h.j.epema@tudelft.nl (D.H.J. Epema).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.05.004

obtain that service. However, the actual number of resources a user can acquire and his waiting time is still an open problem [8]. The latest version of some of the Grid workflow management systems, like Pegasus, VGrADS [9], and ASKALON [10] also supports running scientific workflows on Clouds.

Workflow scheduling is the problem of mapping each task to a suitable resource and of ordering the tasks on each resource to satisfy some performance criterion. Since task scheduling is a well-known NP-complete problem [11], many heuristic methods have been proposed for homogeneous [12] and heterogeneous distributed systems like Grids [13–16]. These scheduling methods try to minimize the execution time (makespan) of the workflows and as such are suitable for community Grids. Most of the current workflow management systems, like the ones mentioned above, use such scheduling methods. However, in Clouds, there is another important parameter other than execution time, i.e., economic cost. Usually, faster resources are more expensive than slower ones, therefore the scheduler faces a time-cost tradeoff in selecting appropriate services, which belongs to the *multi-criteria optimization problems* family. A taxonomy of the multi-criteria workflow scheduling on the Grid can be found in [17], followed by a survey and analysis of the existing scheduling algorithms and workflow management systems.

In our previous work [18], we proposed a QoS-based workflow scheduling algorithm on utility Grids, called the *Partial Critical Paths* (PCP) algorithm, which aims to create a schedule that minimizes the total execution cost of a workflow, while satisfying a user-defined deadline. The PCP algorithm comprises two main phases: Deadline Distribution, which distributes the overall deadline of the workflow across individual tasks, and Planning, which schedules each task on the cheapest service that can execute the task before its subdeadline. However, there are three significant differences between the current commercial Clouds and the utility Grid model for which we devised the PCP algorithm. The first difference is the on-demand (dynamic) resource provisioning feature of the Clouds, which enables the scheduling algorithm to determine the type and the amount of required resources, while in utility Grids, there are predetermined and limited resources with restricted available time slots. This property gives the illusion of unlimited resources to the Cloud users [7]. The second distinction is the (approximately) homogeneous bandwidth among services of a Cloud provider, versus the heterogeneous bandwidth between service providers in the utility Grids. The third (and most important) difference is the pay-as-you-go pricing model of current commercial Clouds which charges users based on the number of the time intervals that they have used. Since the time interval is usually long (e.g., one hour in Amazon EC2) and the user is completely charged for the last time interval even if he uses only a small fraction of it, the scheduling algorithm should try to utilize the last interval as much as possible. Considering these differences, we adapt the PCP algorithm and propose two novel workflow scheduling algorithms for IaaS Clouds, which are called the IaaS Cloud-Partial Critical Paths (IC-PCP) and the IaaS Cloud-Partial Critical Path with Deadline Distribution (IC-PCPD2). IC-PCPD2 is a two-phase algorithm similar to the original PCP, but the deadline distribution and the planning phases are modified to adapt to the Cloud environment. On the other hand, IC-PCP is a one-phase algorithm which uses a similar policy to the deadline distribution phase of the original PCP algorithm, except that it actually schedules each workflow task, instead of assigning a subdeadline to it. Because there is no competitive algorithm in this area, we have compared our algorithms with a modified version of the Loss scheduling algorithm [19] (i.e., IC-Loss) through simulation. The simulation results on five well-known scientific workflow show that the performance of IC-PCP algorithm is better than that of the IC-PCPD2 and IC-Loss algorithms.

The remainder of the paper is organized as follows. Section 2 describes our system model, including the application model, the Cloud model, and the objective function. The proposed scheduling algorithms are explained in Section 3. A performance evaluation is presented in Section 4. Section 5 reviews related work and Section 6 concludes.

2. Scheduling system model

The proposed scheduling system model consists of an application model, an IaaS Cloud model, and a performance criterion for scheduling. An application is modeled by a directed acyclic graph $G(T, E)$, where T is a set of n tasks $\{t_1, t_2, \ldots, t_n\}$, and E is a set of dependencies. Each dependency $e_{i,j} = (t_i, t_j)$ represents a precedence constraint which indicates that task t_i should complete executing before task t_j can start. In a given task graph, a task without any parent is called an *entry task*, and a task without any child is called an *exit task*. As our algorithm requires a single entry and a single exit task, we always add two dummy tasks t_{entry} and t_{exit} to the beginning and the end of the workflow, respectively. These dummy tasks have zero execution time and they are connected with zero-weight dependencies to the actual entry and exit tasks.

Our Cloud model consists of an IaaS provider which offers virtualized resources to its clients. In particular, we assume that the service provider offers a computation service like Amazon Elastic Compute Cloud (EC2) [20] which we can use to run the workflow tasks, and a storage service like Amazon Elastic Block Store (EBS) [21] which can be attached to the computation resources as a local storage device to provide enough space for the input/output files. Furthermore, the service provider offers several computation services $S = \{s_1, s_2, \ldots, s_m\}$ with different QoS parameters such as CPU type and memory size, and different prices. Higher QoS parameters, e.g., a faster CPU or more memory, mean higher prices. We assume that there is no limitation on using each service, i.e., the user can launch any number of instances from each computation service at any time. Some service providers may limit the total number of allocated services to a user, e.g., Amazon currently limits its common users to a maximum of 20 instances of EC2 services. This does not affect our algorithms, unless it limits the number of instances of a particular service. In other words, if the total number of required services (determined by the scheduling algorithm) is more than the maximum limitation of the service provider, then the workflow cannot execute on that provider with the required QoS.

The pricing model is based on a pay-as-you-go basis similar to the current commercial Clouds, i.e., the users are charged based on the number of *time intervals* that they have used the resource, even if they have not completely used the last time interval. We assume each computation service s_i, has a cost c_i for each time interval. Besides, $\mathrm{ET}(t_i, s_j)$ is defined as the execution time of task t_i on computation service s_j. All computation and storage services of a service provider are assumed to be in the same physical region (such as Amazon Regions), so the average bandwidth between the computation services is roughly equal. With this assumption, the data transfer time of a dependency $e_{i,j}$, $\mathrm{TT}(e_{ij})$, only depends on the amount of data to be transferred between corresponding tasks, and it is independent of the services that execute them. The only exception is when both tasks t_i and t_j are executed on the same instance of a computation service, where $\mathrm{TT}(e_{ij})$ becomes zero. Furthermore, the internal data transfer is free in most real Clouds (like Amazon), so the data transfer cost is assumed to be zero in our model. Of course, the service provider charges the clients for using the storage service based on the amount of allocated volume, and possibly for the number of I/O transactions from/to outside the Cloud. Since these parameters have no effect on our scheduling algorithm, we do not consider them in the model.

The last element in our model is the performance criterion which is to minimize the execution cost of the workflow, while completing the workflow before the user specified deadline.

3. IaaS cloud partial critical paths algorithms

In this section, first we compare our previously proposed PCP algorithm for the utility Grid model with the new algorithms, then define some basic definitions, and finally elaborate on the IC-PCP and IC-PCPD2 algorithms, including their time complexities.

3.1. Original PCP vs. IC-PCP and IC-PCPD2

The original PCP scheduling algorithm which was proposed for the utility Grid model [18], has two main phases: *Deadline Distribution* and *Planning*. In the *Deadline Distribution* phase, the overall deadline of the workflow is distributed over individual tasks. In order to do this, first the algorithm finds the overall critical path of the workflow and calls the path assigning algorithm to distribute the deadline among the critical nodes. We proposed three different path assigning policies: *Optimized*, *Decrease Cost*, and *Fair*. After this distribution, each critical task (i.e., a task which is located on the critical path) has a subdeadline which can be used to compute a subdeadline for all of its predecessors in the workflow. Therefore, the PCP algorithm carries out the same procedure for all critical tasks, i.e., it considers a critical task in turn as an exit node and its subdeadline as a deadline, creates a *partial critical path* that ends in that critical task and that leads back to an already assigned task (i.e., a task which has already been given a subdeadline) and then calls path assigning algorithm. This procedure continues recursively until all tasks are successfully assigned a subdeadline. Finally, the *Planning* algorithm schedules the workflow by assigning each task to the cheapest service which meets its subdeadline.

IC-PCPD2 is a two-phase algorithm which has a similar structure to the PCP with two main differences. First, the three path assigning policies have been replaced with a single new policy to adapt to the new pricing model. Second, the planning phase is modified in a way that to schedule a task, it first tries to utilize the remaining time of the existing instances of the computation services, and if it fails, then it considers launching a new instance to execute the task before its subdeadline. On the other hand, IC-PCP is a one-phase algorithm which uses a strategy similar to the deadline distribution phase of the PCP, but instead of assigning subdeadlines to the tasks of a partial critical path, it tries to actually schedule them by finding an (existing or a new) instance of a computation service which can execute the entire path before its latest finish time. In the following sections, we elaborate on the details of the IC-PCP and IC-PCPD2 algorithms.

3.2. Basic definitions

In our scheduling algorithms, we have two notions of the start times of tasks, the earliest start time computed before scheduling the workflow, and the actual start time which is computed after the tasks are scheduled. The Earliest Start Time of each *unscheduled* task t_i, EST (t_i), is defined as follows:

$$\mathrm{EST}\left(t_{entry}\right) = 0 \tag{1}$$

$$\mathrm{EST}\left(t_i\right) = \max_{t_p \in t_i's\ parents} \left\{\mathrm{EST}\left(t_p\right) + \mathrm{MET}\left(t_p\right) + \mathrm{TT}\left(e_{p,i}\right)\right\}$$

where the Minimum Execution Time of a task t_i, MET(t_i), is the execution time of task t_i on a service $s_j \in S$ which has the minimum ET(t_i, s_j) between all available services. Note that MET(t_{entry}) and MET(t_{exit}) equal zero. Accordingly, the Earliest Finish Time of an unscheduled task t_i, EFT (t_i), can be defined as follows:

$$\mathrm{EFT}\left(t_i\right) = \mathrm{EST}\left(t_i\right) + \mathrm{MET}\left(t_i\right). \tag{2}$$

Algorithm 1 The IC-PCP Scheduling Algorithm

```
1: procedure ScheduleWorkflow(G(T, E), D)
2:     determine available computation services
3:     add t_entry, t_exit and their corresponding dependencies to G
4:     compute EST(t_i), EFT(t_i) and LFT(t_i) for each task in G
5:     AST(t_entry) ← 0, AST(t_exit) ← D
6:     mark t_entry and t_exit as assigned
7:     call AssignParents(t_exit)
8: end procedure
```

Also, we define Latest Finish Time of an unscheduled task t_i, LFT (t_i), as the latest time at which t_i can finish its computation such that the whole workflow can finish before the user defined deadline, D. It can be computed as follows:

$$\mathrm{LFT}\left(t_{exit}\right) = D \tag{3}$$

$$\mathrm{LFT}\left(t_i\right) = \min_{t_c \in t_i's\ children} \left\{\mathrm{LFT}\left(t_c\right) - \mathrm{MET}\left(t_c\right) - \mathrm{TT}\left(e_{i,c}\right)\right\}.$$

The Selected Service for each scheduled task t_i, $SS\left(t_i\right) = s_{j,k}$, is defined as the service selected for processing t_i during scheduling, where $s_{j,k}$ is the kth instance of service s_j. Besides, the Actual Start Time of t_i, AST (t_i), is defined as the actual start time of t_i on its selected service. These attributes will be determined after scheduling.

The most important concept in our algorithms is the concept of *Partial Critical Path (PCP)*. However, to define PCP, first we have to introduce the notions of *assigned node* and *Critical Parent*. In the IC-PCP algorithm, an *assigned node* is a node which has already been assigned to (scheduled on) a service, i.e., its selected service has been determined. The definition of assigned node in the IC-PCPD2 algorithm is a bit different: it is a node that a subdeadline has already been assigned to it. Having the definition of an assigned node, we can define the concept of Critical Parent.

The Critical Parent of a node t_i is the *unassigned* parent of t_i that has the latest data arrival time at t_i, that is, it is the parent t_p of t_i for which $\mathrm{EFT}\left(t_p\right) + \mathrm{TT}\left(e_{p,i}\right)$ is maximal.

Now, we can define the main concept of PCP.

The Partial Critical Path of a node t_i is:

i empty if t_i does not have any unassigned parents.
ii consists of the Critical Parent t_p of t_i and the Partial Critical Path of t_p if has any unassigned parents.

3.3. IC-PCP scheduling algorithm

Algorithm 1 shows the pseudo-code of the overall IC-PCP algorithm for scheduling a workflow. After some initialization in lines 2–4, the algorithm sets the actual start time of the dummy nodes t_{entry} and t_{exit}, and marks them as assigned. Note that $AST\left(t_{exit}\right)$ is set to the user's deadline which enforces its parents, i.e., the actual exit nodes of the workflow, to be finished before the deadline. Finally, the *AssignParents* procedure is called for t_{exit} in line 7 which schedules all unassigned parents of its input node. As it has been called for t_{exit}, it will schedule all workflow tasks. We elaborate on this procedure in the next section.

3.3.1. Parents assigning algorithm

The pseudo-code for *AssignParents* is shown in Algorithm 2. This algorithm receives an assigned node as input and schedules all of its unassigned parents before the start time of the input node itself (the while loop from line 2–14). First, *AssignParents* finds the *Partial Critical Path* of the input node (lines 3–7). Note that in the first call of this algorithm, it begins with t_{exit} and follows back the critical parents until it reaches t_{entry}, and so it finds the overall real critical path of the workflow graph.

Then the algorithm calls procedure *AssignPath* (line 8), which receives a path (an ordered list of nodes) as input, and schedules

Algorithm 2 Parents Assigning Algorithm

```
1: procedure AssignParents(t)
2:     while (t has an unassigned parent) do
3:         PCP ← null, t_i ← t
4:         while (there exists an unassigned parent of t_i) do
5:             add CriticalParent(t_i) to the beginning of PCP
6:             t_i ← CriticalParent(t_i)
7:         end while
8:         call AssignPath(PCP)
9:         for all (t_i ∈ PCP) do
10:            update EST and EFT for all successors of t_i
11:            update LFT for all predecessors of t_i
12:            call AssignParents(t_i)
13:        end for
14:    end while
15: end procedure
```

Algorithm 3 Path Assigning Algorithm

```
1: procedure AssignPath(P)
2:     s_{i,j} ← the cheapest applicable existing instance for P
3:     if (s_{i,j} is null) then
4:         launch a new instance s_{i,j} of the cheapest service s_i which can finish each
       task of P before its LFT
5:     end if
6:     schedule P on s_{i,j} and set SS(t_i), AST(t_i) for each t_i ∈ P
7:     set all tasks of P as assigned
8: end procedure
```

the whole path on the cheapest service which can finish each task before its latest finish time. We elaborate on this procedure in the next section. When a task is scheduled, the ESTs and the EFTs of its successors and the LFTs of its predecessors may change, for this reason the algorithm updates these values for all tasks of the path in the next loop. After that, the algorithm starts to schedule the parents of each node on the partial critical path, from the beginning to the end of the path, by calling *AssignParents* recursively (lines 9–13).

3.3.2. Path assigning algorithm

The *AssignPath* algorithm receives a path as input and schedules all of its tasks on a single instance of a computation service with the minimum price which can finish each task before its latest finish time. Since all tasks of the path are scheduled on the same instance, the data transfer times between them become zero, but the data transfer time from the outside tasks should be considered. Algorithm 3 shows the pseudo-code of the AssignPath algorithm. First, the algorithm tries to schedule the entire path on the cheapest *applicable* existing instance for the input path (line 2). An instance is called *applicable* for a path if it satisfies two conditions:

i The path can be scheduled on the instance such that each task of the path is finished before its latest finish time.
ii The new schedule uses (a part of) the extra time of the instance, which is the remaining time of the last time interval of that instance.

There are three important notes about finding an applicable instance. First, to schedule a path on an existing instance, the algorithm considers two cases. If the instance executes one of the children of the last task of the path, it tries to schedule the entire path right before that child by shifting forward the children and its successors on that instance. Otherwise, the algorithm considers both, to schedule the path before the start time of the first task of the instance, and after the finish time of the last task of the instance. Second, the cost of using an existing instance for the input path is equal to the total cost of the new time intervals which are added to the instance for executing the input path, i.e., the cost of using the extra time of an existing instance is zero. Third, if an existing instance executes the parent of a task belongs to the input path, the data transfer time between them becomes zero when considering that instance for execution of the input path.

If the algorithm cannot find an applicable existing instance, it considers to launch a new instance of the cheapest service which can executes each task of the path before its latest finish time (lines 3–5). Finally, the input path is scheduled on the selected instance and all of its tasks are marked as assigned.

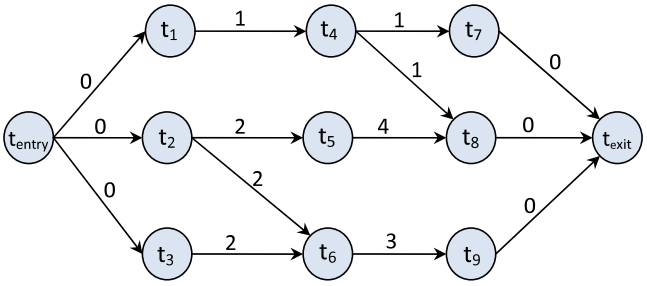


Fig. 1. A sample workflow.

3.3.3. Time complexity

To compute the time complexity of the IC-PCP algorithm, suppose that ScheduleWorkflow has received a workflow $G(T, E)$ as input with n tasks and e dependencies. As G is a directed acyclic graph, the maximum number of dependencies is $\frac{(n-1)(n-2)}{2}$, so we can assume that $e \simeq O(n^2)$. The first part of the algorithm is to initialize the parameters of each workflow task (i.e., EST, EFT and LFT) which requires a forward and a backward processing of all workflow nodes and edges, so its time complexity equals $O(n+e) \backsimeq O(n^2)$.

Then the procedure AssignParents is called which is a recursive procedure that schedules the input workflow. To compute the time complexity of this procedure, we consider its overall actions instead of entering into its details (i.e., computing partial critical paths and calling AssignPath). Overall, the AssignParents procedure schedules each workflow task only once, and updates the parameters of its successors and predecessors. To schedule a task (inside a partial critical path), first the algorithm tries all existing instances to find an applicable one, and then examines available computation services to launch a new instance. Suppose that the maximum number of existing instances is m, and the number of available computation services is constant and small enough to be ignored. Therefore, the time complexity of scheduling all workflow tasks is $O(m \cdot n)$. However, the maximum number of existing instances is at most equal to n (when each task is scheduled on a distinct instance), so the time complexity of the scheduling part becomes $O(n^2)$. On the other hand, each task has at most $n-1$ successors and predecessors, so the time complexity of the updating part for all workflow tasks is $O(n^2)$. Consequently, the overall time complexity of the AssignParents algorithm is $O(n^2)$ which is also the time complexity of the IC-PCP algorithm.

3.3.4. An illustrative example

In order to show how the algorithm works, we trace its operation on the sample graph shown in Fig. 1. The graph consists of nine tasks from t_1 to t_9, and two dummy tasks, t_{entry} and t_{exit}. The number above each arc shows the estimated data transfer time between the corresponding tasks. We assume there are three available computation services (S_1, S_2 and S_3) which can be used to execute the workflow tasks. Table 1 shows the execution times of the workflow tasks on each computation service. The time interval of the computation services is assumed to be 10, and the cost of each time interval is equal to 5 for S_1, 2 for S_2, and 1 for S_3. Finally, let the overall deadline of the workflow be 30.

Table 1
Available computation services and their execution times for the workflow of Fig. 1.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
S_1	2	5	3	4	3	4	5	3	5
S_2	5	12	5	6	8	8	8	6	8
S_3	8	16	9	10	11	11	11	8	14

When we call the IC-PCP scheduling algorithm, i.e., Algorithm 1, for the sample workflow of Fig. 1, it first computes the Earliest Start Time (EST), the Earliest Finish Time (EFT), and the Latest Finish Time (LFT) for each task of the workflow by assigning them to their fastest service. The initial value of these parameters are shown in Table 2 in front of the *Initial* row. Then, the algorithm sets some parameters, and finally, it calls the main procedure of the algorithm, *AssignParents* which will be discussed now.

First, the procedure *AssignParents* (Algorithm 2) is called for task t_{exit}. As this task has three parents, the while loop in line 2 will be executed three times, which we call Step 1 to Step 3, and each one will be discussed separately. The new values of EST, EFT, and LFT of the workflow tasks for each step are given in the related row in Table 2. Note that the appearance of a star mark (*) in front of a cell shows that the value of that cell has been changed over the previous step. Besides, instead of allocating a separate row to the Actual Start Time of a task, we indicate its value in the EST row after the task is scheduled. Therefore, the EST row shows the value of AST for the scheduled tasks.

Step 1: At first, the *AssignParents* algorithm follows the critical parent of t_{exit} to find its partial critical path, which is t_2–t_6–t_9. Then it calls *AssignPath* (Algorithm 3) to schedule these tasks. Since there is no existing instance, the algorithm decides to launch a new instance of S_2 (or $S_{2,1}$), which is the cheapest service that can finish these tasks before their LFTs. Remember that the data transmission time between these tasks becomes zero. The next step is to update the EST of all unassigned successors of these three tasks, i.e., t_5 and t_8, and also the LFT of their unassigned predecessors, i.e., t_3. These changes are shown in Table 2 in front of the Step 1 row. The final step is to call *AssignParents* recursively for all tasks on the path. Since tasks t_2 and t_9 have no unassigned parents, we just consider calling of *AssignParents* for task t_6 in Step 1.1.

Step 1.1: When *AssignParents* is called for task t_6, it first finds the partial critical path of this task, which is t_3. Then it calls *AssignPath* for this task, which decides to launch a new instance of S_3 (or $S_{3,1}$) to execute task t_3. Since t_3 has no unassigned child or parent, Step 1 finishes here.

Step 2: Now, back to task t_{exit}, the *AssignParents* tries to find the next partial critical path of this task, which is t_5–t_8. Then it calls *AssignPath*, which schedules these two tasks on a new instance of S_2 (or $S_{2,2}$). The tasks have no unassigned successor, but the algorithm updates the LFT of their unassigned predecessors, which are t_1 and t_4. Finally, the algorithm calls *AssignParents* for all tasks on the path, t_5 has no unassigned parent, so we just consider t_8 in Step 2.1.

Step 2.1: When *AssignParents* is called for task t_8, it finds the partial critical path for it, which is t_1–t_4, and then calls *AssignPath* which computes the cheapest schedule of these tasks as assigning them to a new instance of S_3 (or $S_{3,2}$). The tasks have no unassigned predecessors, but the algorithm updates the EST of t_4's child, which is t_7. As t_1 and t_4 have no unassigned parents, Step 2 stops.

Step 3: In the final step, *AssignParents* finds the last partial critical path of t_{exit}, which is t_7. Then, *AssignPath* schedules t_7 on the cheapest applicable existing instance $S_{3,2}$, and since there is no unassigned parent or child, the algorithm stops.

The selected instance for each task is shown in the last row of Table 2. Table 3 shows some information about the instances which are launched by IC-PCP. The total execution time of the workflow is 29, and its total cost equals 14.

3.4. IC-PCPD2 scheduling algorithm

The IC-PCPD2 algorithm has two phases: Deadline Distribution and Planning. In the deadline distribution phase, the overall deadline of the workflow is distributed over individual tasks, and in the planning phase each task is scheduled on an instance of a computation service according to its assigned subdeadline. The overall structure of the algorithm is the same as Algorithm 1, except that after calling *AssignParents* (line 7), a new line should be inserted to call *Planning*. However, in the IC-PCPD2 algorithm, the *AssignParents* procedure tries to assign subdeadlines to all unassigned parents of its input node, using a similar strategy to the IC-PCP algorithm. The actual scheduling is carried out by the *Planning* procedure which schedules each task on the cheapest instance that can execute the task before its subdeadline. Furthermore, instead of initializing the actual start times of t_{entry} and t_{exit} in line 5, their subdeadlines should be initialized to the same values, i.e., subdeadline(t_{entry}) $\leftarrow 0$ and subdeadline(t_{exit}) $\leftarrow D$. We will elaborate on the details of these procedures in the next sections.

Table 2
The values of EST, EFT and LFT for each step of running IC-PCP on the sample workflow of Fig. 1.

Tasks		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
	EST	0	0	0	3	7	7	8	14	14
Initial	EFT	2	5	3	7	10	11	13	17	19
	LFT	19	16	16	24	23	22	30	30	30
	EST	0	0	0	3	14*	12*	8	21*	20*
Step 1	EFT	2	12*	3	7	17*	20*	13	24*	28*
	LFT	19	14*	12*	24	23	22	30	30	30
	EST	0	0	0	3	14	12	8	21	20
Step 1.1	EFT	2	12	9*	7	17	20	13	24	28
	LFT	19	14	12	24	23	22	30	30	30
	EST	0	0	0	3	14	12	8	22*	20
Step 2	EFT	2	12	9	7	22*	20	13	28*	28
	LFT	18*	14	12	23*	24*	22	30	30	30
	EST	0	0	0	8*	14	12	19*	22	20
Step 2.1	EFT	8*	12	9	18*	22	20	24*	28	28
	LFT	13*	14	12	23	24	22	30	30	30
	EST	0	0	0	8	14	12	18*	22	20
Step 3	EFT	8	12	9	18	22	20	29*	28	28
	LFT	18	14	12	23	24	22	30	30	30
Selected instance		$S_{3,2}$	$S_{2,1}$	$S_{3,1}$	$S_{3,2}$	$S_{2,2}$	$S_{2,1}$	$S_{3,2}$	$S_{2,2}$	$S_{2,1}$

Table 3
Instances which are launched by IC-PCP to execute the workflow of Fig. 1.

	Start time	Stop time	Duration	Total cost	Assigned tasks
$S_{2,1}$	0	28	28	6	$t_2,\ t_6,\ t_9$
$S_{2,2}$	14	28	14	4	$t_5,\ t_8$
$S_{3,1}$	0	9	9	1	t_3
$S_{3,2}$	0	29	29	3	$t_1,\ t_4,\ t_7$

3.4.1. Parents assigning algorithm

The *AssignParents* algorithm is exactly the same as Algorithm 2. The main difference is when the procedure *AssignPath* is called, it assigns subdeadlines to all unassigned parents of its input node, instead of scheduling them on a service. Another difference is the concept of *assigned node*, which is a node that has already assigned a subdeadline.

3.4.2. Path assigning algorithm

The *AssignPath* algorithm receives a path, $P = \{t_1, \ldots, t_k\}$, as input and assigns subdeadlines to each of its nodes. In order to do this, it tries to fairly distribute the path subdeadline among all tasks of the path in proportion to their minimum execution time. The path subdeadline, PSD, is defined as the difference between the latest finish time of the last task, and the earliest start time of the first task on the path. The subdeadline of each task t_i can be computed as follows:

$$\mathrm{PSD} = \mathrm{LFT}(t_k) - \mathrm{EST}(t_1) \tag{4}$$

$$\mathrm{subdeadline}(t_i) = \frac{\mathrm{EFT}(t_i) - \mathrm{EST}(t_1)}{\mathrm{EFT}(t_k) - \mathrm{EST}(t_l)} \times \mathrm{PSD}. \tag{5}$$

The *AssignPath* algorithm first computes the subdeadline of each task of the input path using Eq. (5), and then marks them as assigned nodes. Due to the simplicity of the algorithm, we omit its pseudo-code.

3.4.3. Planning algorithm

The planning algorithm, which is shown in Algorithm 4, tries to select the cheapest available instance for each task which can finish the task before its subdeadline. At each stage, it selects a ready task, i.e., a task all of whose parents have already been scheduled, and schedules it on the cheapest *applicable* existing instance. The definition of an *applicable* instance for a task is almost similar to its definition for a path in Section 3.3.2, i.e., an instance is applicable for a task if it can be executed on that instance before its subdeadline, and uses (a part of) the extra time of the instance. Again, the cost of using the extra time of an existing instance is considered to be zero. If the algorithm cannot find an applicable existing instance, it launches a new instance of the cheapest service which can finish the task before its subdeadline, and schedules the task on the new instance. This procedure continues until all tasks are scheduled.

3.4.4. Time complexity

To compute the time complexity of the IC-PCPD2 algorithm, we should consider its main parts, i.e., parameter initializing (ESTs, EFTs and LFTs), subdeadline assigning, and planning. The time complexity of the first part is $O(n^2)$ similar to the IC-PCP algorithm. The subdeadline assigning part is similar to the AssignParents procedure in the IC-PCP algorithm, i.e., it consists of two parts: assigning a subdeadline to each task, and updating its successors and predecessors. The subdeadline assigning part is $O(n)$ in this procedure, but the updating part is $O(n^2)$, so the final time complexity is $O(n^2)$. Finally, the planning part schedules each task by considering all existing instances for it, so its time complexity equals $O(m \cdot n) \simeq O(n^2)$. Therefore, the overall time complexity of the IC-PCPD2 algorithm is $O(n^2)$.

Algorithm 4 Planning Algorithm

```
1: procedure PLANNING(G(T, E))
2:     Queue ← t_entry
3:     while (Queue is not empty) do
4:         t ← delete first task from Queue
5:         s_{i,j} ← the cheapest applicable existing instance for P
6:         if (s_{i,j} is null) then
7:             launch a new instance s_{i,j} of the cheapest service s_i which can finish t
   before its subdeadline
8:         end if
9:         schedule t on s_{i,j} and set SS(t) and AST(t)
10:        for all (t_c ∈ children of t) do
11:            if (all parents of t_c are scheduled) then
12:                add t_c to Queue
13:            end if
14:        end for
15:    end while
16: end procedure
```

4. Performance evaluation

In this section we present our simulations of the IC-PCP and IC-PCPD2 algorithms.

4.1. Experimental workflows

To evaluate a workflow scheduling algorithm, we should measure its performance on some sample workflows. There are two ways to choose these sample workflows:

i Using a random DAG generator to create a variety of workflows with different characteristics such as number of tasks, depth, width, etc.
ii Using a library of realistic workflows which are used in the scientific or business community.

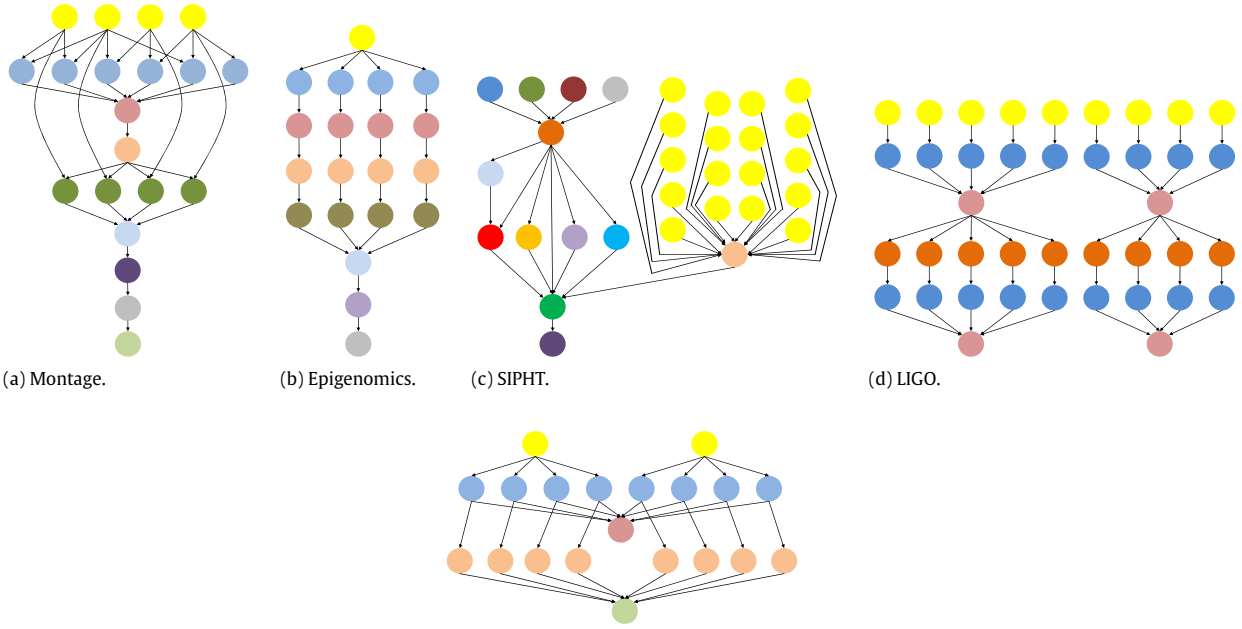
Although the latter seems to be a better choice, unfortunately there is no such a comprehensive library available to researchers. One of the preliminary works in this area was done by Bharathi et al. [22]. They study the structure of five realistic workflows from diverse scientific applications, which are:

- Montage: astronomy
- CyberShake: earthquake science
- Epigenomics: biology
- LIGO: gravitational physics
- SIPHT: biology.

They provide a detailed characterization for each workflow and describe its structure and data and computational requirements. Fig. 2 shows the approximate structure of a small instance of each workflow. It can be seen that these workflows have different structural properties in terms of their basic components (pipeline, data aggregation, data distribution and data redistribution) and their composition. For each workflow, the tasks with the same color are of the same type and can be processed with a common service.

Furthermore, using their knowledge about the structure and composition of each workflow and the information achieved from actual execution of these workflows on the Grid, Bharathi et al. developed a workflow generator, which can create synthetic workflows of arbitrary size, similar to the real world scientific workflows. Using this workflow generator, they create four different sizes for each workflow application in terms of total number of tasks. These workflows are available in DAX (Directed Acyclic Graph in XML) format from their website,[1] from which we choose three sizes for our experiments, which are: Small (about 30 tasks), Medium (about 100 tasks) and Large (about 1000 tasks).

[1] https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.



(a) Montage. (b) Epigenomics. (c) SIPHT. (d) LIGO.

(e) CyberShake.

Fig. 2. The structure of five realistic scientific workflows [22].

4.2. The competitive algorithm

To the best of our knowledge, there is no other workflow scheduling algorithm which solves exactly the same problem (see Section 5). Therefore, we adapted a workflow scheduling algorithm on utility Grids to the IaaS Clouds environment, in order to compare it with our algorithms. This algorithm is called Loss and has been proposed by Sakellariou et al. [19]. It has a different performance criterion: minimizing the execution time under budget constraint. The Loss algorithm consists of two phases: first it tries to find an initial schedule for the input workflow with minimum execution time, and then it refines the initial schedule until its budget constraint is satisfied. In the first phase, it uses a well-known makespan minimization algorithm, called HEFT [13]. HEFT is a list scheduling algorithm [12] which selects tasks according to the descending order of their upward rank, and schedules them on the resource which can finish them as early as possible. The upward rank (or bottom level) of a task t_i is the length of a longest path from t_i to the exit node [12]. If the cost of the initial schedule is less than the user-defined budget, the algorithm stops. Otherwise, it tries to refine the initial schedule by assigning a new resource to a task which has the minimum loss in execution time for the highest cost decrease. For this reason, it computes a weight for assignment of each task to each resource as follows:

$$\text{LossWeight}(t_i, r) = \frac{T_{\text{new}} - T_{\text{old}}}{C_{\text{old}} - C_{\text{new}}}$$

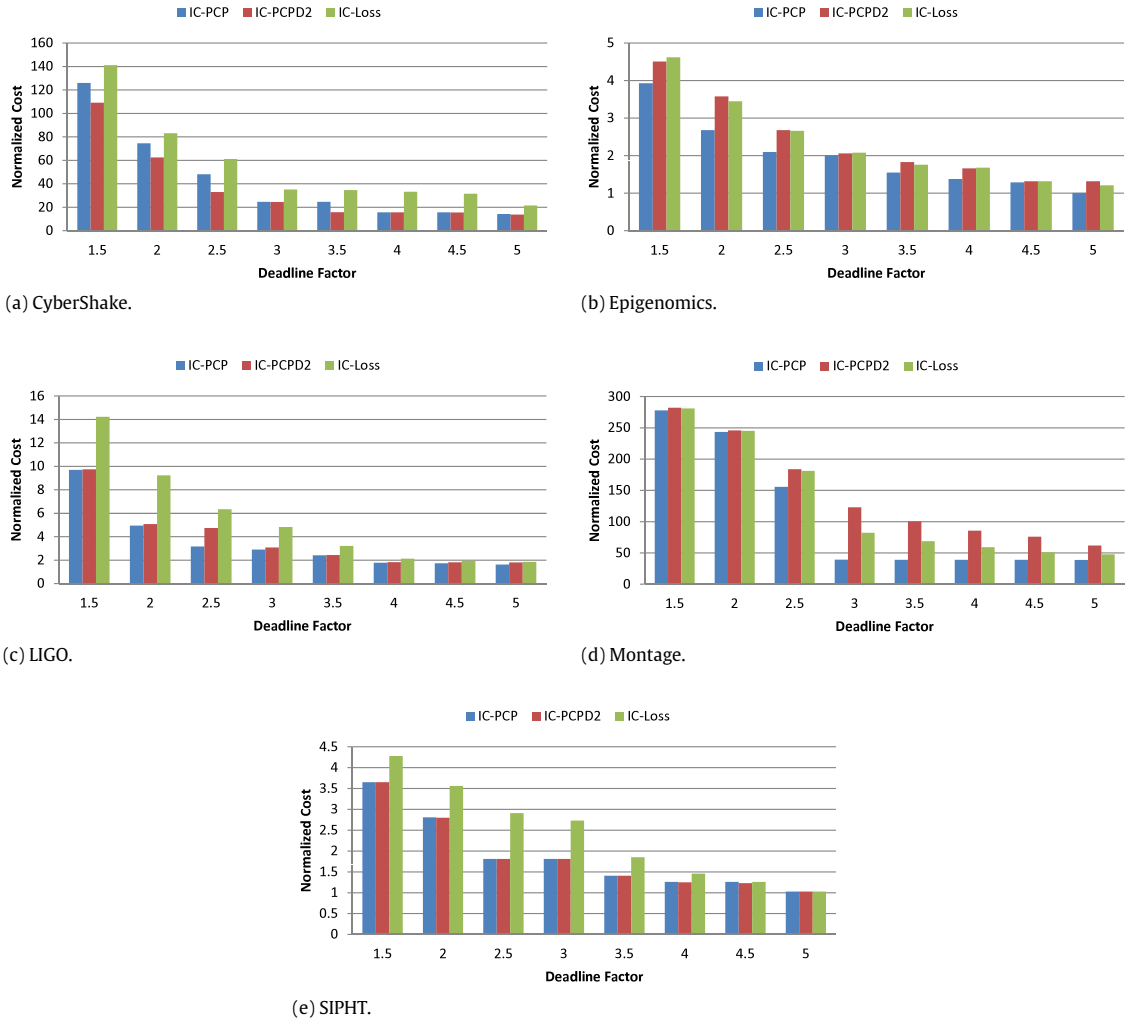
where T_{old} and C_{old} are the execution time and cost of running task t_i in the initial schedule, respectively, and T_{new} and C_{new} are the execution time and cost of running task t_i on resource r. The algorithm chooses the combination of task and resource which has the smallest value of LossWeight, and reschedules the task on that resource. This process continues until the total execution cost falls under the user-defined budget.

We compared our algorithms to IaaS Cloud-Loss (IC-Loss), which is a variation of the Loss algorithm for the deadline-constrained workflow scheduling in the IaaS Clouds. To devise IC-Loss, first we have to adapt the HEFT algorithm for the IaaS Cloud environment. For this reason, we just consider instances of the fastest computation service for scheduling the input workflow with HEFT. Besides, in the resource selection stage of the HEFT algorithm, we use a similar method to the planning phase of IC-PCPD2, i.e., the algorithm tries to find the instance which has the minimum finish time for the selected task among existing instances which their extra time can be used by that task, plus a new instance of the fastest computation service. Subsequently, we should adapt the refinement phase of the Loss algorithm. Unfortunately, we cannot consider each task individually, because rescheduling an individual task on a cheaper service may increase the total execution cost (if it cannot use the extra time of the new instance). Therefore, IC-Loss tries to reschedule all tasks of an instance to a cheaper existing or new instance, in order to reduce the total execution cost. For this reason, the algorithm computes LossWeight(T_I, J), where T_I is the set of tasks which are currently scheduled on an existing instance I, and J is an existing or a new cheaper instance than I. The other parameters are defined accordingly, e.g., T_{new} is the total execution time of running all tasks of T_I on instance J. The rest of the algorithm is similar to the original Loss, i.e., it chooses the instance which has the minimum LossWeight, and schedules all of its tasks on the cheaper instance. This process continues as long as the total execution time does not exceed the user-defined deadline.

4.3. Experimental setup

For our experiments, we assume a Cloud environment which consists of a service provider, offers 10 different computation services (similar to Amazon EC2 services), with different processor speeds and different prices. The processor speeds are selected randomly such that the fastest service is roughly five times faster than the slowest one, and accordingly, it is roughly five times more expensive. The average bandwidth between the computation services is set to 20 MBps which is the approximate average bandwidth between the computation services (EC2) in Amazon [23].

Another important parameter of the experiments is the time interval. Most of the current commercial Clouds, such as Amazon, charge users based on a long time interval equal to one hour. As a result, these Clouds do not exactly meet the goal of the pay-as-you-go pricing model, because the user has to pay for the whole



(a) CyberShake.

(b) Epigenomics.

(c) LIGO.

(d) Montage.

(e) SIPHT.

Fig. 3. The Normalized Cost of scheduling workflows with IC-PCP, IC-PCPD2 and IC-Loss with the time interval equal to 1 h.

last time interval, even if he has used a fraction of it. Therefore, the users prefer shorter time intervals in order to pay close to what they have really used. Today, a few service providers support short time intervals, for example CloudSigma[2] has a burst pricing model that its time interval is five minutes. However, it is possible that more service providers support short time intervals in the future, in order to be competitive. To evaluate the impact of short and long time intervals on our algorithms, we consider two different time intervals in the experiments: a long one equal to one hour, and a short one equal to five minutes.

Since a large set of workflows with different attributes is used, it is important to normalize the total cost of each workflow execution. For this reason, first we define the *Cheapest schedule* as scheduling all workflow tasks on a single instance of the cheapest computation service, according to their precedence constraints. Now we can define the Normalized Cost (NC) of a workflow execution as follows:

$$\mathrm{NC} = \frac{\textit{total schedule cost}}{C_C} \tag{6}$$

where C_C is the cost of executing the same workflow with the *Cheapest* strategy.

Finally, to evaluate the IC-PCP scheduling algorithm, we need to assign a deadline to each workflow. To do this, first we define the *Fastest schedule* as scheduling each workflow task on a distinct instance of the fastest computation service, while all data transmission times are considered to be zero. Despite the Cheapest schedule, the Fastest schedule of a workflow is not a real schedule (unless the workflow is linear). Therefore, the makespan of the Fastest schedule of a workflow, which is denoted by M_F, is just a lower bound for the makespan of executing that workflow. In order to set deadlines for workflows, we define the *deadline factor* α, and we set the deadline of a workflow to the time of its arrival plus $\alpha \cdot M_F$. Since the problem has no solution for $\alpha = 1$, we let α ranges from 1.5 to 5 in our experiments, with a step length equal to 0.5.

4.4. Experimental results

Fig. 3 shows the cost of scheduling large workflows with IC-PCP, IC-PCPD2 and IC-Loss with the time interval equal to one hour. It should be mentioned that all algorithms have successfully scheduled all workflows before their assigned deadlines, even in the case of tight deadlines, i.e., $\alpha = 1.5$. A quick look at Fig. 3 shows that IC-PCP outperforms IC-Loss in all cases, but IC-PCPD2 has a better performance than IC-Loss for the LIGO, CyberShake, and SIPHT workflows. For the Montage workflow, IC-PCPD2 has the worst performance between all three algorithms. Furthermore, Fig. 3 shows that IC-PCP has a better overall performance over IC-PCPD2. The performance of IC-PCP is notably better than IC-PCPD2 for the Montage and Epigenomics workflows, and it

[2] www.cloudsigma.com.

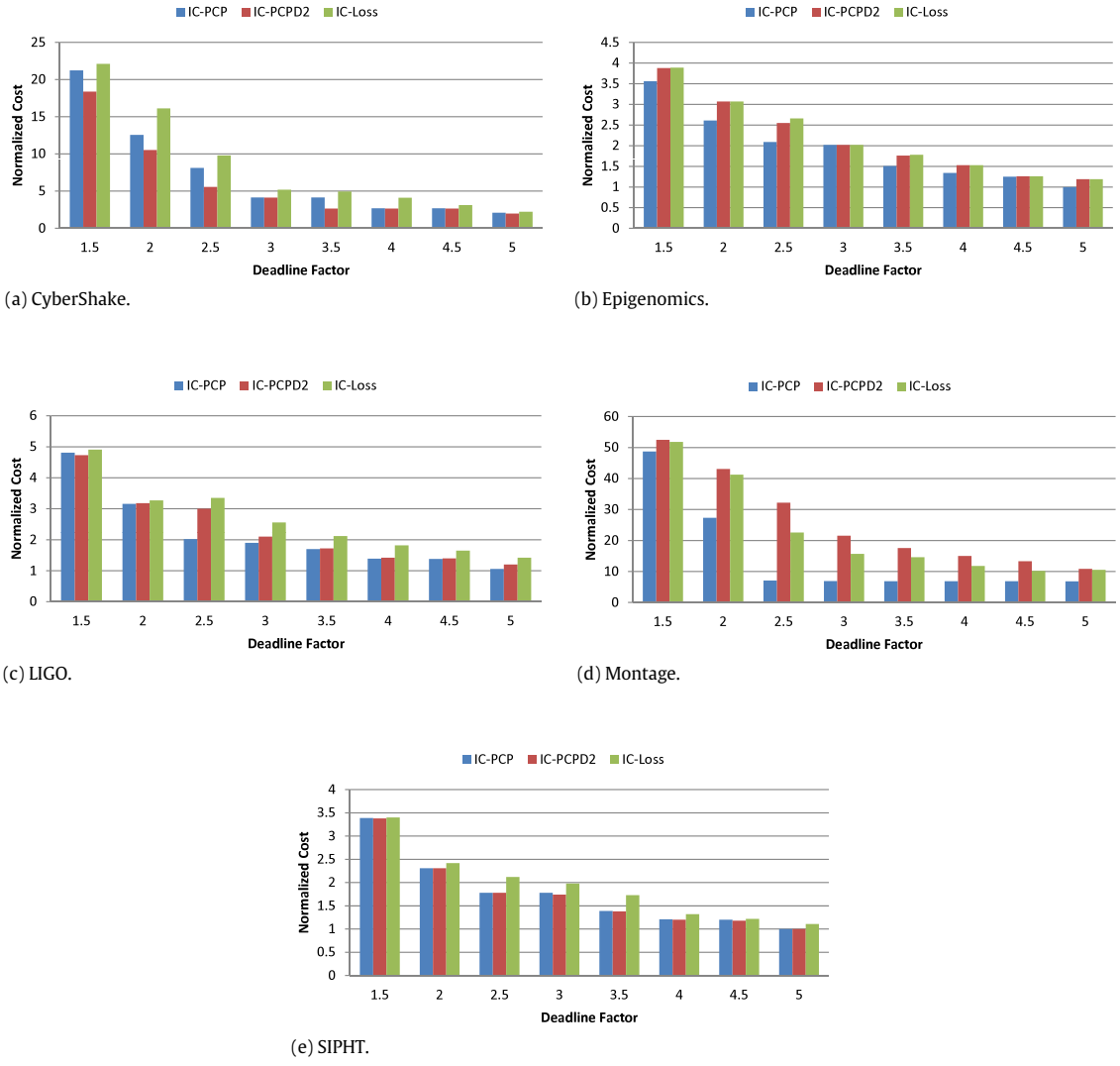


Fig. 4. The Normalized Cost of scheduling large workflows with IC-PCP, IC-PCPD2 and IC-Loss with the time interval equal to 5 min.

is slightly better for the LIGO workflow. In the case of SIPHT, both algorithms have an approximately similar performances. CyberShake is the only workflow for which IC-PCPD2 has a better performance.

Fig. 3(d) and (a) show that the value of NC is very high for the Montage and CyberShake workflows, e.g., it is about 277.9 for scheduling Montage with the IC-PCP algorithm when $\alpha = 1.5$. To find the reason, consider the Montage structure in Fig. 2(a). It can be seen that there are many tasks in the second row of the workflow, with a short execution time about 10 seconds on the fastest service. When the deadline is tight (e.g. $\alpha = 1.5$), the only way to finish all of these tasks before the single task in the third row is to assign a distinct instance to each of them (or to a few of them). Therefore, the scheduling algorithm has to launch many instances of the computation service, while just a small fraction of their time interval is used. This causes the overall cost of the schedule to increase drastically, especially in the case of a long time interval. The reason is similar for the CyberShake workflow.

Fig. 4 shows the cost of scheduling large workflows with IC-PCP, IC-PCPD2 and IC-Loss with the time interval equal to five minutes. The overall results are almost similar to the previous experiments with a long time interval, except the value of NC is decreased as expected. The decrease in NC is not significant for the Epigenomics and SIPHT workflows, because both of them have time consuming tasks for which the value of the time interval has no important impact. However, for the Montage and CyberShake workflows, the value of NC is considerably decreased (about one fifth), because their small tasks can use a larger fraction of a short time interval. This is also the case for the LIGO workflow, and its NC is decreased by about one half.

We also repeated the same experiments for the small and medium size workflows and the overall structure of the results is approximately similar to the large workflows. The main difference between the results of the different workflow sizes is the value of NC. To investigate the impact of the workflow size on the value of NC, we consider the results of the IC-PCP algorithm for different workflow sizes, when $\alpha = 1.5$ and the time interval is equal to 1 h. For the CyberShake workflow, the value of NC is decreased from 126.07 for the large size instance to 88.5 for the medium size, and 56.25 for the small size instance. The reason is when the number of small tasks of the CyberShake is decreased, the number of allocated computation instances is also decreased. This is also the case for the Montage workflow. Conversely, the value of NC for Epigenomics is increased from 3.92 for the large instance, to 4.9 for the small instance. This is also the case for the SIPHT workflow, where NC is increased from 3.64 for the large one to 4.12 for the medium one, and to 4.91 for the small one. The value of NC remains almost constant for different sizes of the LIGO workflow.

Finally, both IC-PCP and IC-PCPD2 algorithms are very fast and their actual computation times are less than 500 ms for the large size workflows.

5. Related work

There are few works addressing workflow scheduling on the Clouds. Hoffa et al. [6] compared the performance of running the Montage workflow on a local cluster, against a science Cloud at the university of Chicago. In a similar work, Juve et al. [7] compared the performance of running some scientific workflows on the NCSA's Abe cluster, against the Amazon EC2. Both of them use Pegasus [3] as the workflow management system to execute the workflows. However, these works are concentrated on the possibility of running workflows on the Cloud environment, and do not consider minimizing the cost of workflow execution. Salehi and Buyya [24] proposed two scheduling policies, i.e., Time Optimization and Cost Optimization, to execute deadline and budget constrained Parameter Sweep Applications on the combination of local resources and public IaaS Clouds. Both policies use a local cluster to execute the input application, while the Time Optimization policy tries to minimize the execution time by hiring resources from a Cloud provider (such as Amazon EC2) within the assigned budget, but the Cost Optimization hires extra Cloud resources only if it cannot meet the user deadline. However, they do not consider workflow applications. Pandey et al. [25] uses Particle Swarm Optimization (PSO) to minimize the execution cost of running workflow application on the Cloud, nevertheless, their Cloud model (e.g., their pricing model) is completely different from ours. Furthermore, meta-heuristic algorithms such as PSO are very time consuming and cannot be used for large workflows. Finally, Xu et al. [26] proposed an algorithm for scheduling multiple workflows with multiple QoS constrains on the Cloud, while our algorithms focus on scheduling a single workflow.

Ostermann et al. [10] consider scheduling workflows on a Grid environment which is capable to lease Cloud resources, whenever needed. They extend the ASKALON workflow management system to utilize Cloud resources, and employ it to run the Wien2k scientific workflow in the Austrian Grid environment which is extended with a local Cloud. Furthermore, in [27] they extend their previous work to minimize the leasing cost of the Cloud resources, while maximizing the reduction of the workflow makespan. The proposed algorithm concentrates on four important issues: when to extend the Grid infrastructure with Cloud resources, how many Cloud resources should be provisioned, when to reschedule some tasks from Cloud to Grid in order to decrease cost or makespan of the workflow, and when to stop Cloud resources to save money. They use a just-in-time scheduling algorithm which schedules each task when it becomes ready to run, unlike our algorithms which are complete full-ahead ones. Besides, in the fourth issue, they also consider the pricing policy of the current commercial Clouds, i.e., if a Cloud resource becomes idle, it is not released until its current time interval gets close to an end, in hope of using it for the future ready tasks. However, both works use Cloud resources in support of running workflow applications on the Grids, unlike our algorithms which are concentrated on the Cloud, itself.

One of the most recent works in this area has been introduced by Byun et al. [28,29]. First, they proposed the Balanced Time Scheduling (BTS) algorithm [28], which schedules a workflow on an on-demand resource provisioning system such as Cloud. The BTS algorithm tries to estimate the minimum number of resources required to execute the input workflow within a user-defined deadline. The algorithm has three main phases: *initialization*, which computes the initial values of some basic parameters, *task placement*, which computes a preliminary schedule for the workflow tasks, and *task redistribution*, which refines the preliminary schedule by relocating tasks in the areas that require more resources. The BTS algorithm has two main shortcomings: (1) it considers the environment as homogeneous, i.e., all resources have the same (or similar) performance per task, (2) it uses a fixed number of resources during the whole workflow execution, i.e., it does not use the elastic and on-demand characteristics of the current resource provisioning systems. In their next effort, Byun et al. proposed the Partitioned Balanced Time Scheduling (PBTS) algorithm [29] which is an extension of the BTS algorithm for elastic resources, i.e., it eliminates the second shortcoming. The PBTS algorithm considers the pricing policy of the current commercial Clouds and their time intervals, which are called time partitions in this algorithm. This algorithm aims to find the minimum number of required resources for each time partition, such that the whole workflow finishes before the user-defined deadline. The PBTS algorithm has three phases: *task selection* phase, which determines the set of tasks for each time partition, *resource capacity estimate and task scheduling* phase, which uses the BTS algorithm to estimate the minimum number of required resources for the selected tasks of each time partition, and *task execution* phase, which actually schedules and executes the workflow tasks on the estimated and provisioned resources. PBTS is the closest algorithm to our work, however, it still suffers from the first shortcoming, i.e., it only considers homogeneous resources, while our algorithms consider a heterogeneous environment.

Moreover, there are some works addressing workflow scheduling on utility Grids, which might be modified to adapt to the Cloud environment. We have already mentioned the Loss algorithm proposed by Sakellariou et al. [19]. They proposed another algorithm, called Gain [19], for the same problem. In this algorithm, they initially assign each task to the cheapest resource, and then try to refine the schedule to shorten the execution time under budget constraint. Their experiments show that Loss has a better performance than Gain.

The Deadline-MDP which is proposed by Yu et al. [30], is one of the most cited algorithms in this area. They divide the workflow into partitions and assigned each partition a subdeadline according to the minimum execution time of each task and the overall deadline of the workflow. Then they try to minimize the cost of each partition execution under its subdeadline constraint. Yuan et al. [31] proposed the DET (Deadline Early Tree) algorithm which is based on a similar method. First, they create a primary schedule as a spanning tree which is called *Early Tree*. Then, they try to distribute the whole deadline among workflow tasks by assigning a time window to each task. This is achieved in two steps: first a dynamic programming approach assigns time windows to critical tasks, then an iterative procedure finds a time window for each non-critical task. Finally, a local optimization method tries to minimize the execution costs according to the dedicated time windows.

Prodan et al. [32] proposed a bi-criteria scheduling algorithm that follows a different approach to the optimization problem of two arbitrary independent criteria, e.g., execution time and cost. In the Dynamic Constraint Algorithm (DCA), the end user defines three important factors: the *primary* criteria, the *secondary* criteria and the *sliding constraint* which determines how much the final solution can vary from the best solution for the primary criterion. The DCA has two phases: (1) primary scheduling finds the preliminary solution which is the best solution according to the primary criterion, (2) secondary scheduling optimizes the preliminary solution for the secondary criterion while keeping the primary criterion within the predefined sliding constraint. The primary scheduling algorithm depends on the type of primary criteria, e.g., HEFT is used for the execution time. For the secondary scheduling, the authors model the problem as a multiple choice knapsack problem and solve it using a heuristic based on a dynamic programming method.

Duan et al. [33] proposed two algorithms based on Game Theory for the scheduling of n independent workflows. The first one called Game-quick, tries to minimize the overall makespan

of all workflows. The second algorithm called Game-cost, tries to minimize the overall cost of all workflows, while meeting each workflow's deadline. Brandic et al. [34] solved the problem of multi-objective workflow scheduling using Integer Programming. To transform the multi-objective problem to a single-objective one, the user should assign a weight to each QoS parameter and the algorithm tries to optimize the weighted sum of the QoS parameters.

Meta-heuristic methods are also used to tackle this problem. Although these methods have a good performance, they are usually more time consuming than the heuristic ones. Yu et al. [35] use the Genetic Algorithm to solve both problems: cost optimization under deadline constraint, and execution time optimization under budget constraint. In another work, Yu et al. [36] use three multi-objective evolutionary algorithms, i.e., NSGAII, SPEA2 and PAES for this problem. In this work, the user can specify his desired deadline and maximum budget and the algorithm proposes a set of alternative trade-off solutions meeting the user's constraints from which he can select the best one according to his needs. In a similar research, Talukder et al. [37] proposed a Multiobjective Differential Evolutionary algorithm which generates a set of alternative solutions for the user to select. Chen and Zhang [38] proposed an Ant Colony Optimization (ACO) algorithm, which considers three QoS parameters, i.e., time, cost and reliability. They enable the users to determine constraints for two of these parameters, and the algorithm finds the optimized solution for the third parameter while meeting those constraints. In [39], the performance of six different algorithms, i.e., Tabu Search, Simulated Annealing, Iterated Local Search, Guided Local Search, Genetic Algorithm and Estimation of Distribution Algorithm are compared together in solving the problem of cost optimization under deadline constraint.

Among the current workflow management systems, VGrADS [9] supports workflow scheduling on the combination of Grid and Cloud systems. VGrADS accepts deadline-driven scientific workflows and initially tries to schedule them on the cheaper Grid resources. In the next phase, it uses the higher-priced Cloud resources (such as Amazon EC2) to meet the user deadline, if necessary. Cloudbus toolkit [40] also includes a workflow engine for running workflows on the Cloud.

6. Conclusions

In this paper we extend our previous algorithm for deadline-constrained workflow scheduling in utility Grid environment, to design two new algorithms, which are called IC-PCP and IC-PCPD2, for the IaaS Cloud environment. The new algorithms consider the main features of the current commercial Clouds such as on-demand resource provisioning, homogeneous networks, and the pay-as-you-go pricing model. The main difference between algorithms is that IC-PCP schedules the workflow in one phase by scheduling each partial critical path on a single instance of a computation service, while IC-PCPD2 is a two-phase algorithm which first distributes the overall deadline on the workflow tasks, and then schedules each task based on its subdeadline. The time complexity of both algorithms is $O(n^2)$, where n is the number of workflow tasks. The polynomial time complexity makes them suitable options for the large workflows. We evaluate our algorithms by comparing their performance on scheduling five synthetic workflows that are based on real scientific workflows with different structures and different sizes, with IC-Loss which is a variation of the Loss [19] scheduling algorithm. The results show that IC-PCP outperforms both, IC-PCPD2 and IC-Loss in most cases. Furthermore, the experiments show that the computation times of the algorithms are very low, less than 500 ms for the large workflows. In the future, we intend to improve our algorithms for the real Cloud environments, such as Amazon. The most important problem in the real environment is the inaccuracy of the estimated execution and transmission times. For this reason, we plan to perform real experiments, find the drawbacks of the algorithms in the real environments, and try to refine the algorithms to improve their performance.

References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: vision, hype and reality for delivering computing as the 5th utility, Future Gener. Comput. Syst. 25 (6) (2009) 599–616.
[2] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: an overview of workflow system features and capabilities, Future Gener. Comput. Syst. 25 (2009) 528–540.
[3] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, D.S. Katz, Pegasus: a framework for mapping complex scientific workflows onto distributed systems, Sci. Prog. 13 (2005) 219–237.
[4] M. Wieczorek, R. Prodan, T. Fahringer, Scheduling of scientific workflows in the askalon Grid environment, SIGMOD Rec. 34 (2005) 56–62.
[5] F. Berman, et al., New Grid scheduling and rescheduling methods in the GrADS project, Int. J. Parallel Program. 33 (2005) 209–229.
[6] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, J. Good, On the use of cloud computing for scientific workflows, in: Fourth IEEE Int'l Conference on e-Science, e-Science 2008, 2008.
[7] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, P. Maechling, Scientific workflow applications on Amazon EC2, in: 5th IEEE International Conference on e-Science, 2009.
[8] E. Deelman, Grids and clouds: making workflow applications work in heterogeneous distributed environments, Int. J. High Perform. Comput. Appl. 24 (2010) 284–298.
[9] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T.M. Huang, K. Thyagaraja, D. Zagorodnov, VGrADS: enabling e-science workflows on Grids and clouds with fault tolerance, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC'09, ACM, New York, NY, USA, 2009, pp. 47:1–47:12.
[10] S. Ostermann, R. Prodan, T. Fahringer, Extending Grids with cloud resource management for scientific computing, in: 2009 10th IEEE/ACM International Conference on Grid Computing, October 2009, pp. 42–49.
[11] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.
[12] Y.K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (4) (1999) 406–471.
[13] H. Topcuoglu, S. Hariri, M. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[14] R. Bajaj, D.P. Agrawal, Improving scheduling of tasks in a heterogeneous environment, IEEE Trans. Parallel Distrib. Syst. 15 (2) (2004) 107–118.
[15] M.I. Daoud, N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 68 (4) (2008) 399–409.
[Online] Available: http://dx.doi.org/10.1016/j.jpdc.2007.05.015.
[16] D. Bozdag, U. Catalyurek, F. Ozguner, A task duplication based bottom–up scheduling algorithm for heterogeneous environments, in: Proc. of the 20th Int'l Parallel and Distributed Processing Symposium, IPDPS'06, April 2006.
[17] M. Wieczorek, A. Hoheisel, R. Prodan, Towards a general model of the multi-criteria workflow scheduling on the Grid, Future Gener. Comput. Syst. 25 (2009) 237–256.
[18] S. Abrishami, M. Naghibzadeh, D.H.J. Epema, Cost-driven scheduling of Grid workflows using partial critical paths, IEEE Trans. Parallel Distrib. Syst. 23 (8) (2012) 1400–1414.
[19] R. Sakellariou, H. Zhao, E. Tsiakkouri, M.D. Dikaiakos, Scheduling workflows with budget constraints, in: S. Gorlatch, M. Danelutto (Eds.), Integrated Research in GRID Computing, CoreGRID Integration Workshop 2005, Selected Papers, in: CoreGRID Series, Springer-Verlag, 2007, pp. 189–202.
[20] Amazon elastic compute cloud (Amazon EC2).
[Online] Available: http://aws.amazon.com/ec2/.
[21] Amazon elastic elastic block store (Amazon EBS).
[Online] Available: http://aws.amazon.com/ebs/.
[22] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, K. Vahi, Characterization of scientific workflows, in: The 3rd Workshop on Workflows in Support of Large Scale Science, 2008.
[23] M.R. Palankar, A. Iamnitchi, M. Ripeanu, S. Garfinkel, Amazon S3 for science Grids: a viable solution? in: Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing, DADC'08, ACM, New York, NY, USA, 2008, pp. 55–64.
[24] M.A. Salehi, R. Buyya, Adapting market-oriented scheduling policies for cloud computing, in: Proceedings of the 10th Int'l Conference on Algorithms and Architectures for Parallel Processing, ICA3PP 2010, 2010, pp. 351–362.

[25] S. Pandey, L. Wu, S. Guru, R. Buyya, A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments, in: 2010 24th IEEE International Conference on Advanced Information Networking and Applications, AINA, 2010, pp. 400–407.
[26] M. Xu, L. Cui, H. Wang, Y. Bi, A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing, in: 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications, 2009, pp. 629–634.
[27] S. Ostermann, R. Prodan, T. Fahringer, Dynamic cloud provisioning for scientific Grid workflows, in: 2010 11th IEEE/ACM International Conference on Grid Computing, GRID, October 2010, pp. 97–104.
[28] E.-K. Byun, Y.-S. Kee, J.-S. Kim, E. Deelman, S. Maeng, Bts: resource capacity estimate for time-targeted science workflows, J. Parallel Distrib. Comput. 71 (6) (2011) 848–862.
[29] E.-K. Byun, Y.-S. Kee, J.-S. Kim, S. Maeng, Cost optimized provisioning of elastic resources for application workflows, Future Gener. Comput. Syst. 27 (8) (2011) 1011–1026. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11000744.
[30] J. Yu, R. Buyya, C.K. Tham, Cost-based scheduling of scientific workflow applications on utility Grids, in: First Int'l Conference on e-Science and Grid Computing, July 2005, pp. 140–147.
[31] Y. Yuan, X. Li, Q. Wang, X. Zhu, Deadline division-based heuristic for cost optimization in workflow scheduling, Inform. Sci. 179 (15) (2009) 2562–2575. Including Special Issue on Computer-supported cooperative work—techniques and applications, The 11th Edition of the International Conference on CSCW in Design.
[32] R. Prodan, M. Wieczorek, Bi-criteria scheduling of scientific Grid workflows, IEEE Trans. Autom. Sci. Eng. 7 (2) (2010) 364–376.
[33] R. Duan, R. Prodan, T. Fahringer, Performance and cost optimization for multiple large-scale Grid workflow applications, in: Proc. of the 2007 ACM/IEEE Conference on Supercomputing, ACM, New York, USA, 2007, pp. 1–12.
[34] I. Brandic, S. Benkner, G. Engelbrecht, R. Schmidt, QoS support for time-critical Grid workflow applications, in: Int'l Conference on e-Science and Grid Computing, IEEE Computer Society, Los Alamitos, USA, 2005, pp. 108–115.
[35] J. Yu, R. Buyya, Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms, Sci. Prog. 14 (3, 4) (2006) 217–230.
[36] J. Yu, M. Kirley, R. Buyya, Multi-objective planning for workflow execution on Grids, in: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, GRID'07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 10–17.
[37] A.K.M.K.A. Talukder, M. Kirley, R. Buyya, Multiobjective differential evolution for scheduling workflow applications on global Grids, Concurrency Computat.: Pract. Exper. 21 (2009) 1742–1756.
[38] W.-N. Chen, J. Zhang, An ant colony optimization approach to Grid workflow scheduling problem with various QoS requirements, IEEE Trans. Syst. Man Cybern. 39 (1) (2009) 29–43.
[39] D.M. Quan, D.F. Hsu, Mapping heavy communication Grid-based workflows onto Grid resources within an SLA context using metaheuristics, Int. J. High Perform. Comput. Appl. 22 (3) (2008) 330–346.
[40] R. Buyya, S. Pandey, C. Vecchiola, Cloudbus toolkit for market-oriented cloud computing, in: Proceedings of the 1st International Conference on Cloud Computing, CloudCom'09, 2009, pp. 24–44.

Saeid Abrishami received the M.Sc. and Ph.D. degrees in Computer Engineering from Ferdowsi University of Mashhad, in 1999 and 2011, respectively. He is currently an Assistant Professor with the Department of Computer Engineering at Ferdowsi University of Mashhad. During the Spring and Summer of 2009 he was a visiting researcher at the Delft University of Technology. His research interests focus on resource management and scheduling in distributed systems, especially in Grids and Clouds.

Mahmoud Naghibzadeh received his M.Sc. degree in Computer Science from the University of Southern California (USC), USA in 1977, and the Ph.D. degree in Computer Engineering from the same university in 1980. Since 1982, he has been with the Computer Engineering Department, Ferdowsi University of Mashhad, where he is currently a Full Professor. During the academic year 1990–1991 he was a visiting scholar at the University of California, Irvine (UCI), USA. Also, in the fall of 2003 he was a visiting professor at Monash University, Australia. His research interests are in the areas of distributed systems, grids, and knowledge engineering.

Dick H.J. Epema received the M.Sc. and Ph.D. degrees in Mathematics from Leiden University, Leiden, the Netherlands, in 1979 and 1983, respectively. Since 1984, he has been with the Department of Computer Science of Delft University of Technology, where he is currently an Associate Professor in the Parallel and Distributed Systems Group. Since 2011, he is also a part-time Full Professor of Decentralized Distributed Systems at Eindhoven University of Technology. During 1987–1988, the fall of 1991, and the summer of 1998, he was a visiting scientist at the IBM T.J. Watson Research Center in New York. In the fall of 1992, he was a visiting professor at the Catholic University of Leuven, Belgium, and in the fall of 2009 he spent a sabbatical at UCSB. His research interests are in the areas of performance analysis, distributed systems, peer-to-peer systems, and grids. He has co-authored over 90 papers in peer-reviewed conferences and journals, was a general co-chair of Euro-Par 2009 and IEEE P2P 2010, and is the general chair of HPDC 2012.