Scientia Iranica D (2012) 19 (3), 680–689

Sharif University of Technology

Scientia Iranica

Transactions D: Computer Science & Engineering and Electrical Engineering

www.sciencedirect.com

Research note

Deadline-constrained workflow scheduling in software as a service Cloud

S. Abrishami[*], **M. Naghibzadeh**

Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, P.O. Box 91755-1111, Iran

Received 27 February 2011; revised 19 June 2011; accepted 14 November 2011

KEYWORDS
Cloud computing;
SaaS 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 in executing scientific applications such as workflows. In this model, the users request for available services according to their desired Quality of Service, and they are charged on a pay-per-use basis. One of the most challenging problems in Clouds is workflow scheduling, i.e., the problem of satisfying the QoS of the user as well as minimizing the cost of workflow execution. In this paper, we propose a new QoS-based workflow scheduling algorithm based on a novel concept called Partial Critical Paths (PCP), which tries to minimize the cost of workflow execution while meeting a user-defined deadline. This algorithm recursively schedules the partial critical paths ending at previously scheduled tasks. The simulation results show that the performance of our algorithm is very promising.

1. Introduction

Cloud computing [1] is the latest emerging trend in distributed computing that delivers hardware infrastructures 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-per-use 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 multi-objective nature of the scheduling problem in Clouds makes it difficult to solve, especially in the case of complex jobs like workflows. This has led most researchers to use time-consuming meta-heuristic approaches, instead of fast heuristic methods. In this paper, we propose a new heuristic algorithm for scheduling workflows in Clouds, and we evaluate its performance on some well-known scientific workflows.

Recently, many researchers have considered the benefits of using Cloud computing for scientific applications [2–4]. Using virtualization technologies, Clouds can offer the users a wide range of services from hardware to the application level. Currently, these services are categorized into three major classes: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS Clouds, like Amazon, provide virtualized hardware and storage on top of which the users can deploy their own applications and services. PaaS Clouds, like Microsoft Azure, provide an application development environment in which the users can implement and run applications on the Cloud. According to [5], there are two types of Cloud, which deliver software applications to the users. The first group offers an entire application as a service to the end users, which can be used without any changes or customization. Examples of these Clouds are Google office automation services like Google Document or Google Calendar. The second group provides rudimentary web services to the users (known as on-demand web services), which can be used to build more complex applications. Xignite and StrikeIron offer web services hosted on a Cloud on a pay per use basis. We have used the second group in our SaaS Cloud model.

Workflows constitute a common model for describing a wide range of scientific applications in distributed systems. Usually, a workflow is described by a Directed Acyclic Graph (DAG) in which each computational task is represented by a

[*] Corresponding author.
E-mail addresses: s-abrishami@um.ac.ir (S. Abrishami), naghibzadeh@um.ac.ir (M. Naghibzadeh).
Peer review under responsibility of Sharif University of Technology.

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 [6], ASKALON [7] and GrADS [8], have designed workflow management systems to define, manage and execute workflows on the Grid. Having been successful on Grids, researchers are now investigating the running of large scientific workflows on Clouds [4]. Furthermore, the latest version of Grid workflow management systems, like Pegasus, VGrADS [9] and ASKALON [10], also support 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. As 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 [13,14]. These scheduling methods try to minimize the execution time (makespan) of the workflows and, as such, are suitable for community Grids. Most current workflow management systems, like the ones above mentioned, use such scheduling methods. However, in Clouds, there are many other potential QoS attributes besides execution time, like reliability, security, availability and so on. Besides, stricter QoS attributes mean higher prices for services. Therefore, the scheduler faces a QoS-cost tradeoff in selecting appropriate services, which belongs to the *multi-objective optimization problems* family. There are several existing approaches to the problem of multi-objective scheduling [15]. One method is to find *pareto optimal* solutions, and let the user select the best schedule according to his requirements. The problem is that the pareto sets are usually very large and hard to examine. Another common method is to assign a weight to each scheduling criterion, and optimize the weighted sum of these criteria. However, in most cases, the weight assignment is not a straightforward process for users. Due to the complexities of the development of a general multi-objective scheduling algorithm, many researchers try to propose bi-criteria scheduling algorithms. In most bi-criteria scheduling algorithms, the user specifies a limitation for one criterion (deadline or budget constraints), and the algorithm tries to optimize the other criterion under this constraint. This is a convenient way for the users to express their requirements, and a helpful method for the researchers to simplify the problem and propose fast and high performance solutions.

In our previous paper, we proposed a QoS-based workflow scheduling algorithm on utility Grids, called the *Partial Critical Paths* (PCP) [16]. In this paper, we propose the SaaS Cloud Partial Critical Paths (SC-PCP) algorithm, which is an extension of the previous one for the SaaS Clouds. The objective function of the SC-PCP algorithm is to create a schedule that minimizes the total execution cost of a workflow, while satisfying a user-defined deadline for the total execution time. First, the SC-PCP algorithm tries to schedule the (overall) critical path of the workflow, such that it is completed before the user deadline, and execution cost is minimized. Then, it finds the *partial critical path* to each scheduled task on the critical path and executes the same procedure in a recursive manner.

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 SC-PCP scheduling algorithm is 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, a Cloud model, and a performance criterion for scheduling. An application is modeled by a directed acyclic graph $w(T, E)$, where T is a set of n tasks $\{t_1, t_2, \dots, t_n\}$, 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 execution 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 end of the workflow, respectively. These dummy tasks have zero execution time and are connected with zero-weight dependencies to the actual entry and exit tasks.

Our Cloud model consists of an SaaS provider which provides web services to its clients and is capable of executing scientific workflows. The service provider offers several services with different QoS for each task of every workflow. We assume that each workflow task, t_i, can be processed by m_i services, $S_i = \{s_{i,1}, s_{i,2}, \dots, s_{i,m_i}\}$, with different QoS attributes. There are many QoS attributes for services, like execution time, cost, reliability, security and so on. In this study, we use the most important ones, execution time and cost, for our scheduling model. Furthermore, the pricing model is *per transaction pricing*, which charges users based on the number of completed transactions. In this context, a completed transaction means a successfully finished workflow task. Some real Clouds, like Xignite and StrikeIron, use this pricing model. The price of a service usually depends on its execution time, i.e. shorter execution times are more expensive. We assume that the services of each task are sorted in an increasing order according to their execution times, i.e. $s_{i,j}$ is faster (and more expensive) than $s_{i,j+1}$. Besides, $ET(t_i, s)$ and $EC(t_i, s)$ are defined as the execution time and the execution cost of processing task t_i on service s, respectively. Finally, we assume that there is no limitation on using services and we can schedule a task on every potential service, at any time. This assumption is based on an important feature of current commercial public Clouds like Amazon, the illusion of unlimited resources [2]. It means that the user can ask for every service of the Cloud whenever he needs it, and he will certainly (or with a very high possibility) obtain that service. Of course, this assumption may fail in the future, when Clouds become more popular.

There is another source of time and money consumption: transferring data between tasks. All tasks are assumed to use a shared storage service (such as Amazon Simple Storage Service [17]) to send and receive the intermediate data. Besides, 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 and the storage service is roughly equal. With this assumption, the data transfer time of a dependency $e_{i,j}$ only depends on the amount of data to be transferred between corresponding tasks, and it is independent of the services which execute them. Therefore, $TT(e_{ij})$ is defined as the data transfer time of a dependency, $e_{i,j}$, independent of the selected service for t_i and t_j. Furthermore, the internal data transfer is free in most real Clouds, 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 are constant for each workflow and they are not influenced by the scheduling algorithm, we do not consider them in the model.

The last element in our model is the performance criterion. In traditional scheduling, users prefer to minimize the completion time (makespan) of their jobs. However, in Clouds, price is an important factor. Generally, a user job has a deadline before which the job must be finished, but earlier completion of the job only incurs more cost to the user. Therefore, our performance criterion is to minimize the execution cost of the workflow, while completing the workflow before the user specified deadline.

3. The Cloud partial critical paths algorithm

In this section, we first elaborate on the SC-PCP scheduling algorithm, and then compute its time complexity.

3.1. Main idea

Critical Path heuristics are widely used in workflow scheduling. The *critical path* of a workflow is the longest execution path between the entry and exit tasks of the workflow. Most of these heuristics try to schedule *critical tasks* (*nodes*), i.e. the tasks belonging to the critical path, first, by assigning them to the resources that process them earliest, in order to minimize the execution time of the entire workflow. Our proposed algorithm is based on a similar heuristic, to schedule the critical nodes first, yet not to minimize the execution time, but to minimize the price of executing the critical path before the user-specified deadline. After scheduling all critical nodes, each of them has a start time which is a deadline for its parent nodes, i.e. its (direct) predecessors in the workflow. So, then we can carry out the same procedure by considering each critical node in turn as an exit node with its start time as a deadline, and creating a *partial critical path* that ends in the critical node and that leads back to an already scheduled node. In the *SaaS Cloud-Partial Critical Paths* (SC-PCP) algorithm, this procedure continues recursively until all tasks are successfully scheduled. In the following sections, we elaborate on the details of the SC-PCP algorithm.

3.2. Basic definitions

For a workflow, a *schedule* is defined as an assignment of services to the workflow tasks. If $SS(t_i)$ denotes the selected service for task t_i, then a schedule of a workflow $w(T, E)$ is defined as:

$$Sched(w) = \{SS(t_i) \mid \forall t_i \in T\ SS(t_i) = s_{i,j_i} \in S_i\}. \quad (1)$$

Having an arbitrary schedule, we define for each task, t_i, its Earliest Start Time, $EST(t_i)$, as the earliest time at which t_i can start its computation. It can be computed as follows:

$$EST(t_{entry}) = 0,$$
$$EST(t_i) = \max_{t_p \in \text{predecessors of } t_i} \{EST(t_p) + ET(t_p, SS(t_p)) + TT(e_{p,i})\}. \quad (2)$$

Accordingly, the Earliest Finish Time of each task, t_i, $EFT(t_i)$, is the earliest time at which t_i can finish its computation, and it is computed as follows:

$$EFT(t_i) = EST(t_i) + ET(t_i, SS(t_i)). \quad (3)$$

Also, we define Latest Finish Time, $LFT(t_i)$, 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:

$$LFT(t_{exit}) = D,$$
$$LFT(t_i) = \min_{t_c \in \text{successors of } t_i} \{LFT(t_c) - ET(t_c, SS(t_c)) - TT(e_{i,c})\}. \quad (4)$$

A *feasible schedule* is defined as a schedule which finishes the workflow before the user's defined deadline. If each task of the workflow is assigned to its fastest service, we call the schedule the *fastest schedule*. According to the Cloud model from the previous section and the above definitions, we can infer four important corollaries that will be used in the algorithm.

Corollary 1. *The fastest schedule for a workflow has the minimum execution time (makespan) amongst all possible schedules, in the proposed Cloud model.*

This can be easily inferred from the specification of the Cloud model. First, the data transfer time between two arbitrary tasks is constant and does not depend on the selected services. So the makespan of the fastest schedule cannot decrease by assigning slower services to two interconnected tasks, in hope of a decrease in the data transfer time between them. Second, as the Cloud services are supposed to be infinite, the makespan of the fastest schedule does not decrease by assigning a slower service to a task, in hope of using the faster service for another urgent task of the same type.

Corollary 2. *If the fastest schedule is not feasible, the scheduling problem has no solution.*

This is an obvious result of Corollary 1.

Corollary 3. *If the selected service of a task, t_i, in an arbitrary schedule is changed, only the ESTs of its successors and the LFTs of its predecessors might change.*

According to Eq. (2), the EST of each task depends on the execution time of its parents. Therefore, changing the selected service of t_i, may change the EST of its children. Subsequently, these changes can be propagated to its indirect successors as well. With a similar reasoning, Eq. (4) shows that the LFT of each task depends on the execution time of its children. Therefore, changing the selected service of t_i, may change the LFT of its parents, and subsequently its indirect predecessors.

Corollary 4. *A schedule is feasible, if and only if $\forall t_i\ EFT(t_i) \leqslant LFT(t_i)$.*

This is obvious from the definitions of EFT and LFT. A schedule is called a feasible schedule for task t_i, if this relation holds for this task.

3.3. The SC-PCP scheduling algorithm

Algorithm 1 shows the pseudo-code of the overall SC-PCP algorithm for scheduling a workflow. After some initialization, the algorithm generates the fastest schedule for the input workflow in line 4. This is a preliminary schedule, which obviously has the highest cost. In the next phases, the algorithm tries to refine this preliminary schedule by changing the selected service of each task, such that the overall execution time extends to the user's deadline, and the cost decreases as much as possible. We define a *scheduled* task as a task whose selected service is finalized and which never changes in the next phases of the algorithm. Obviously, all tasks are still *unscheduled* in the preliminary schedule.

Then, the algorithm computes the ESTs and the LFTs of all tasks according to the preliminary schedule. In line 7, the algorithm checks if a solution exists, according to Corollary 2. After that, the dummy nodes, t_{entry} and t_{exit}, are marked as scheduled and finally the *ScheduleParents* procedure is called for t_{exit} in line 11. This procedure schedules all unscheduled parents of its input node. As it has been called for t_{exit}, it will schedule all workflow tasks.

3.4. The parents scheduling algorithm

The pseudo-code for *ScheduleParents* is shown in Algorithm 2. This algorithm receives a scheduled node as input and schedules all its unscheduled parents before the start time of the input node itself (the while loop from line 2 to 14). First, *ScheduleParents* try to find the *Partial Critical Path* of unscheduled nodes ending at its input node and starting at one of its predecessors that has no unscheduled parent. For this reason, it uses the concept of *Critical Parent*.

Definition 1. The Critical Parent of node t_i is the unscheduled 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 $EST(t_p) + ET(t_p, SS(t_p)) + TT(e_{p,i})$ is maximal.

We will now define the fundamental concept of the SC-PCP algorithm.

```
1:procedure ScheduleWorkflow($w(T, E), D$)
2:      determine available services $S_i$ for each task type in $w$
3:      add $t_{entry}, t_{exit}$ and their corresponding dependencies to $w$
4:      generate the fastest schedule
5:      compute $EST(t_i)$ for each task in $w$ according to Eq. (2)
6:      compute $LFT(t_i)$ for each task in $w$ according to Eq. (4)
7:      if (the fastest schedule is not feasible) then
8:          return(null)
9:      end if
10:     mark $t_{entry}$ and $t_{exit}$ as scheduled
11:     call ScheduleParents($t_{exit}$)
12:end procedure
```

Algorithm 1. The SC-PCP scheduling algorithm.

```
1: procedure ScheduleParents($t$)
2:     while($t$ has an unscheduled parent) do
3:         $PCP \leftarrow null, t_i \leftarrow t$
4:         while (there exists an unscheduled parent of $t_i$) do
5:             add $CriticalParent(t_i)$ to the beginning of $PCP$
6:             $t_i \leftarrow CriticalParent(t_i)$
7:         end while
8:         call $SchedulePath(PCP)$
9:         for all ($t_i \in PCP$) do
10:            update $EST$ for all successors of $t_i$
11:            update $LFT$ for all predecessors of $t_i$
12:            call $ScheduleParents(t_i)$
13:        end for
14:    end while
15: end procedure
```

Algorithm 2. Parents scheduling algorithm.

Definition 2. The Partial Critical Path of node t_i is:

(i) Empty if t_i does not have any unscheduled parents.
(ii) Consists of critical parent t_p of t_i and the Partial Critical Path of t_p if has any unscheduled parents.

Algorithm 2 begins with the input node and follows the critical parents until it reaches a node that has no unscheduled parent, to form a partial critical path (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 complete workflow graph.

Then, the algorithm calls procedure *SchedulePath* (line 8), which receives a path (an ordered list of nodes) as input, and schedules each node on the path, such that it can complete before its latest finish time, and the total execution cost of the path is minimized. We elaborate on this procedure in the next sub-section. As the *SchedulePath* probably changes the selected services of some tasks on the path, the ESTs of their successors and the LFTs of their predecessors may change (according to Corollary 3). 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 *ScheduleParents* recursively (lines 9–13).

3.5. The path scheduling algorithm

The *SchedulePath* algorithm receives a path as input and tries to find a schedule for its tasks that minimizes the total cost of the path and finishes each task before its latest finishes time. We propose three different policies for scheduling a path as follows.

Optimized policy. In this policy, we try to find the cheapest schedule that can finish the tasks of the path before their latest finish time. Since the problem of finding the optimal schedule for an ordered list of tasks, or, more precisely, a linear workflow is also an NP-complete problem, there is no polynomial time algorithm to solve it. Fortunately, this problem can be formulated as an extension of a classic problem, known as the *Multiple Choice Knapsack Problem* (*MCKP*) [18].

Problem 1. Given l classes, $N_1, \ldots, N_l$, of items to pack in some knapsack of capacity W. Each item, $j \in N_i$, has profit p_{ij} and weight w_{ij}. The problem is to choose, at most, one item from each class, such that the profit sum is maximized without having the weight sum exceed W. The MCKP is formulated as follows:

maximize:

$$z = \sum_{i=1}^{l} \sum_{j \in N_i} p_{ij} x_{ij},$$

subject to:

$$\sum_{i=1}^{l} \sum_{j \in N_i} w_{ij} x_{ij} \leqslant W,$$

$$\sum_{j \in N_i} x_{ij} = 1, \quad i = 1, \ldots, l$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \ldots, l, \; j \in N_i.$$

To formulate our scheduling problem as an MCKP problem, consider a path of l tasks, $\{t_1, t_2, \ldots, t_l\}$, to be scheduled. The set, S_i, of available services for each task corresponds to the set, N_i, of available items for each class of items. The total execution time of each task on an available service, defined as $TT(e_{i-1,i}) + ET(t_i, s_{i,j})$ except for the first task which is $ET(t_1, s_{1,j})$, corresponds to the weight of an item in each class of items, w_{ij}. Similarly, the execution cost of each task on an available service, $EC(t_i, s_{i,j})$, corresponds to the profit of an item in each class of items, c_{ij}. Finally, the *path deadline*, defined as $LFT(t_l) - EST(t_1)$, corresponds to the total weight of the knapsack, W, and the total execution cost of the path corresponds to the total profit of the selected items. The only difference is that the scheduling problem tries to minimize cost, instead of maximizing profit.

Formulating the path scheduling problem as an instance of MCKP has the benefit of many efficient exact and approximated algorithms which have been proposed over the decades. One of the first exact methods of solving MCKP used *Dynamic Programming* [18], which we have modified to solve the scheduling problem.

Suppose $\{t_1, t_2, \dots, t_l\}$ is the input path to the algorithm, constituting l tasks. Obviously, the whole path should be completed before $LFT(t_l)$. Let $C_k(d)$ represents the minimum execution cost of scheduling tasks t_1 to t_k, before the sub-deadline, d. For $C_1(d)$, we have:

$$C_1(d) = \begin{cases} +\infty \quad d = EST(t_1), \dots, EFT(t_1) - 1 \\ \min\limits_{s \in S_1}\{EC(t_1, s), EST(t_1) + ET(t_1, s) \leqslant d\} \\ \quad d = EFT(t_1), \dots, LFT(t_1) \\ +\infty \quad d = LFT(t_1) + 1, \dots, LFT(t_l). \end{cases} \tag{5}$$

Now, $C_k(d)$ can be recursively computed as follows:

$$\begin{aligned} C_k(d) &= +\infty, \\ &\quad d = EST(t_1), \dots, EFT(t_k) - 1, \\ C_k(d) &= \min_{s \in S_k}\{C_{k-1}(d - ET(t_k, s) - TT(e_{k-1,k})) + EC(t_k, s), \\ &\qquad EST(t_k) + ET(t_k, s) \leqslant d\}, \\ &\quad d = EFT(t_k), \dots, LFT(t_k), \\ C_k(d) &= +\infty, \\ &\quad d = LFT(t_k) + 1, \dots, LFT(t_l). \end{aligned} \tag{6}$$

The cheapest schedule can be found by computing $C_l(LFT(t_l))$. This policy is shown in Algorithm 3. The time complexity of the algorithm is $O(l.m.d)$, where m is the maximum number of service types for an arbitrary task, and d is the path deadline, i.e. $d = LFT(t_l) - EST(t_1)$. It is called pseudo-polynomial, because it is polynomial in the numeric value of d, but it is exponential in the length of d, i.e. its number of digits.

```
1: procedure SchedulePath($path$)
2:     for all tasks $t_k \in path$ do
3:         for $d = EST(t_1)$ to $EFT(t_k) - 1$ do
4:             $C[k, d] = \infty$
5:         end for
6:         for $t = EFT(t_k)$ to $LFT(t_k)$ do
7:             if ($t_k$ is the first task on the path) then
8:                 compute $c[1, t]$ according to Eq. (5)
9:             else
10:                compute $c[k, t]$ according to Eq. (6)
11:            end if
12:        end for
13:        for $t = LFT(t_k) + 1$ to $LFT(t_l)$ do
14:            $c[k, t] = \infty$
15:        end for
16:    end for
17:    find the optimal schedule based on $c[l, LFT(l)]$
18:    set EST, LFT and SS based on the optimal schedule
19:    mark all tasks of the $path$ as scheduled
20: end procedure
```

Algorithm 3. Optimized path scheduling algorithm.

This algorithm can efficiently solve the MCKP in many cases. However, the most effective exact algorithm for the MCKP is based on the *Branch and Bound* approach [18]. These algorithms usually find the optimal solution for a relaxed version of the problem, e.g. linear programming relaxation, which lets $0 \leqslant x_{ij} \leqslant 1$, and use it as an upper bound for the original problem. Using this upper bound, they eliminate partial solutions whose upper bound is less than the current best solution. Finally, there are some polynomial time *approximation algorithms* which try to find an inexact (approximate) solution with a bounded worst-case relative error, denoted by ϵ. It means $P - P^* \leqslant \epsilon P^*$, where P^* is the optimal solution for the problem, and P is the solution found by the approximation algorithm. To find some references to these algorithms, see [18].

Decrease cost policy. This policy is based on a *Greedy* approach that tries to approximate the previous (optimized) policy, i.e. it tries to find a good (but not necessarily optimal) solution with a polynomial time complexity. Remember that in the preliminary schedule, the algorithm assigned the fastest service to each task of the workflow, including those tasks on the input path. Therefore, the current schedule of the path is the most expensive one. This policy tries to decrease the cost by assigning cheaper (and therefore slower) services to the tasks, without exceeding the LFT of any task. To determine which task should be scheduled to a cheaper service, we first compute the Cost Decrease Ratio, CDR, which is defined as follows:

$$CDR(t_i) = \frac{EC(t_i, cs) - EC(t_i, ns)}{ET(t_i, ns) - ET(t_i, cs)}, \tag{7}$$

where *cs* is the current service that has been assigned to task t_i and *nr* is the next slowest service to the current one for t_i. The CDR of task t_i shows how much cheaper in execution it will be in taking one unit of time longer. Then, task t^* is selected, such that it has the maximum CDR and it is *replaceable*, i.e. scheduling it on the next slower service generates a feasible schedule. Finally, the current service of t^* is changed to the next slower service. Algorithm 4 shows this policy.

The time complexity of this algorithm is better than the previous one. Suppose that the maximum number of available service types for a task is m, and l is the path length. The most time consuming part is the repeat-until loop. In the worst case, all tasks can try all their available services, so this loop can be run at most $l.m$ times. As this loop has a nested ForAll loop, which is run l times, the ultimate time complexity is $O(l^2.m)$.

```
1: procedure SchedulePath($path$)
2:     compute $CDR(t_i)$ for each task of the path according to Eq. (7)
3:     repeat
4:         $t^* \leftarrow null$
5:         for all ($t_i \in path$) do
6:             if ($CDR(t_i) > CDR(t^*)$ and $t_i$ is replaceable) then
7:                 $t^* \leftarrow t_i$
8:             end if
9:         end for
10:        if ($t^*$ is not null) then
11:            $SS(t^*) \leftarrow$ next slower service
12:            update the EST and the LFT of other tasks on the path
13:            update $CDR(t^*)$
14:        end if
15:    until ($t^*$ is null)
16:    mark all tasks of the $path$ as scheduled
17: end procedure
```

Algorithm 4. Decrease cost path scheduling algorithm.

Fair policy: This policy, like the previous one, tries to decrease cost by scheduling each task on a cheaper service, while preserving the schedule as feasible. The main difference is that it tries to follow a fair policy in selecting tasks for rescheduling. For this reason, the policy starts from the first task towards the last task, and substitutes the selected service to the next slower service, provided that the schedule remains feasible. This procedure continues iteratively until no substitution can be made. The policy has been shown in Algorithm 5. In the worst case, the repeat-until loop can be executed m times, so the time complexity of the algorithm is $O(l.m)$.

3.6. Time complexity

To compute the time complexity of our proposed algorithm, suppose that ScheduleWorkflow has received a workflow, $w(T, E)$, as input with n tasks and e dependencies. We assume that the maximum number of available services for an arbitrary task is m, the length of the longest path between entry and exit tasks is l, and D is the user-defined deadline. As w 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 most time consuming part of the ScheduleWorkflow is the *ScheduleParents* algorithm. Nevertheless, we first compute the time complexity of other (main) parts of the algorithm as follows:

```
1: procedure SchedulePath($path$)
2:     repeat
3:         for all ($t_i \in Path$) do
4:             if (scheduling $t_i$ on the next service is feasible) then
5:                 $SS(t_i) \leftarrow$ next slower service
6:                 update the EST and the LFT of other tasks on the path
7:             end if
8:         end for
9:     until (no change is done)
10:    set EST, LFT and SS according to $best$ for all tasks $\in path$
11:    mark all tasks of the $path$ as scheduled
12: end procedure
```

Algorithm 5. Fair path scheduling algorithm.

- Line 4 (generating preliminary schedule): $O(n)$,
- Line 5 (computing ESTs): $O(n + e) = O(n^2)$,
- Line 6 (computing LFTs): $O(n + e) = O(n^2)$.

The *ScheduleParents* algorithm is a recursive procedure. In the first place, it is called for the exit task and then it calls itself for all the workflow tasks. The algorithm has a while loop (lines 2–14) that processes all incoming dependencies of each node (task), so it will process all workflow dependencies. Inside the while loop, first, it computes the partial critical path whose time complexity is $O(l)$. Then, it calls *SchedulePath* whose time complexity depends on the selected policy. Having the time complexity of our three *SchedulePath* policies, the time complexity of *ScheduleParents* will be $O(e.l + e.(l.m.D))$, $O(e.l + e.(l^2.m))$ and $O(e.l + e.(l.m))$, respectively. Obviously, the first part of the time complexity is dominated by the second part, so it can be omitted. If we replace e with n^2, and considering the fact that $l \ll D$, then we have the final time complexities as $O(n^2.m.D)$, $O(n^2.l^2.m)$ and $O(n^2.l.m)$, which is also the time complexity of the whole SC-PCP algorithm. Note that *ScheduleParents* also updates the EST of all successors, and the LFT of all predecessors of each node after scheduling. In the worst case, a node has $n - 1$ successors and predecessors, so the time complexity of updating ESTs and LFTs for all nodes will be $O(n^2)$, which can be omitted against the bigger part of the time complexity.

Now, let us consider parameter l and that how big it can be. As we defined before, l is the length of the longest path between entry and exit tasks, so its maximum value can be n, i.e. when we have a linear workflow. In this case, the time complexity of *ScheduleParents* will be $O(n^2.m.D)$, $O(n^4.m)$ and $O(n^3.m)$, respectively. On the other hand, if we consider real workflows (like realistic workflows we have used in the evaluation section), we see that for many of them, the value of l cannot take such a big value. The value of l shows, in some way, the number of stages in a workflow, especially for the structured workflows. When we consider large workflows, it can be seen that the number of tasks (nodes) is high, but the depth of the workflow is a reasonable value. In other words, when a specific workflow gets larger, the number of tasks increases, but the number of stages (length) remains the same. This is the case for the realistic workflows we have used in the evaluation section. Although we cannot say it is a general rule for workflows, it is quite reasonable for many realistic workflows. Having this assumption, we can consider l as a constant in the time complexity computation. Furthermore, the number of services provided by a service provider is limited, so the value of m can be considered as a constant too. Having these assumptions, the time complexity of *the SC-PCP* will be $O(n^2.D)$, $O(n^2)$ and $O(n^2)$, respectively.

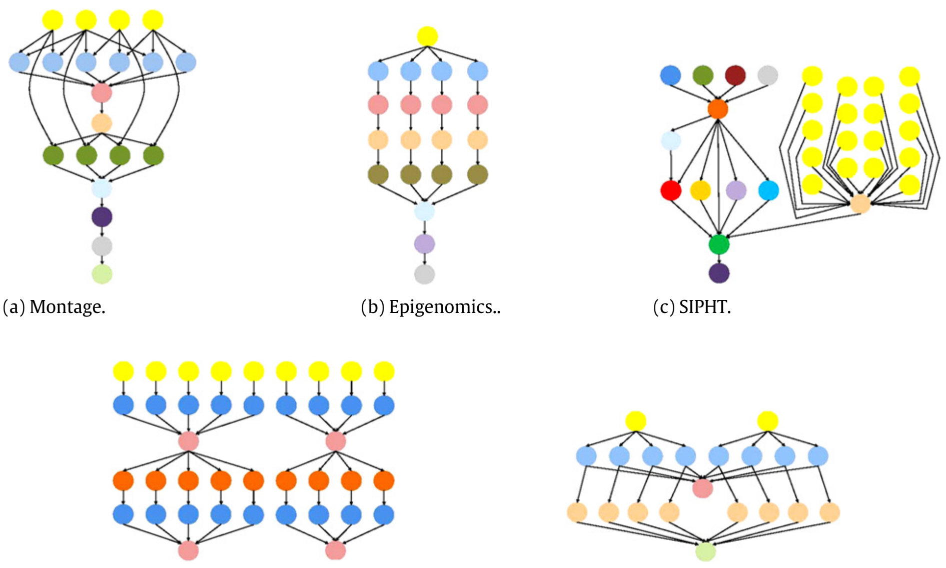
4. Performance evaluation

In this section, we will present our simulations of the Cloud Partial Critical Paths algorithm.

4.1. Experimental setup

A Java-based simulator is developed to simulate the Cloud environment for our experiments. The simulated Cloud model consists of a service provider, and offers services to execute each task type of the five experimental workflows, e.g. 9 different services for the Montage workflow. Each task type is supported by 10 services with different execution times and costs. The execution time of each task type on its fastest service is set to be equal to its average execution time, explained in [19], and execution times on slower services are selected randomly. The execution cost of each service is also selected randomly, such that a faster service costs more than a slower one. To control the range of execution times and costs, they are selected, such that the fastest service for each task type is approximately 10 times faster than the slowest one, and accordingly it is roughly 10 times more expensive. The average bandwidth between the computation services and the storage service is set to 20 MBps, which is the approximate average bandwidth between the storage service (S3) and the computation service (EC2) in Amazon [20].

To evaluate our workflow scheduling algorithm, we have used a library of realistic workflows that are introduced by Bharathi et al. [19]. They study the structure of five realistic workflows from diverse scientific applications, which are: Montage (astronomy), CyberShake (earthquake science), Epigenomics (biology), LIGO (gravitational physics) and SIPHT (biology). Figure 1 shows the approximate structure of a small instance of each workflow. For each workflow, the tasks with the same color are of the same type and can be processed with a common service. Furthermore, Bharathi et al. developed a workflow generator, which can create synthetic workflows of arbitrary size, similar to real world scientific workflows. Using this workflow generator, they create four



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

(d) LIGO. (e) CyberShake.

Figure 1: The structure of five realistic scientific workflows [19]. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

different sizes for each workflow application in terms of total number of tasks. These workflows are available from their website (https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator) from which we have chosen three sizes for our experiments. They are: Small (about 30 tasks), Medium (about 100 tasks) and Large (about 1000 tasks).

Finally, the SC-PCP algorithm is compared with the Deadline-MDP, one of the most cited algorithms in this area, which has been proposed by Yu et al. [21]. This algorithm has been designed for utility Grids, but it can be used in the Cloud environment too. They divided the workflow into partitions and assigned each partition a sub-deadline, according to the minimum execution time of each task and the overall deadline of the workflow. Then, they tried to minimize the cost of each partition execution under its sub-deadline constraint.

4.2. Experimental results

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 assigning each task of the workflow to its cheapest service. Now, we can define the Normalized Cost (NC) of a workflow execution as follows:

$$NC = \frac{\text{total schedule cost}}{C_C}, \tag{8}$$

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

To evaluate our SC-PCP scheduling algorithm, we need to assign a deadline to each workflow. Clearly, this deadline must be greater than, or equal to, the makespan of scheduling the same workflow with the *Fastest* strategy. 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 $\alpha \cdot M_F$, where M_F is the makespan of executing the same workflow with the *Fastest* strategy. In our experiments, we let α range from 1 to 5.

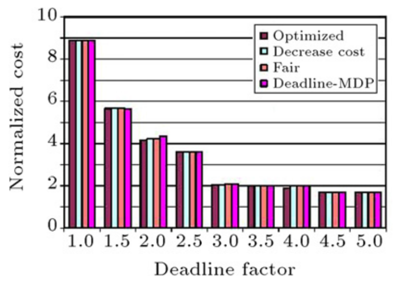
Table 1: Average cost decrease percentage of the SC-PCP (optimized policy) over the deadline-MDP.

	Small	Medium	Large
CyberShake	1.39	1.79	21.34
Epigenomics	16.08	17.79	19.75
LIGO	9.45	9.84	9.26
Montage	18.43	25.15	41.66
SIPHT	18.5	16.99	15.96

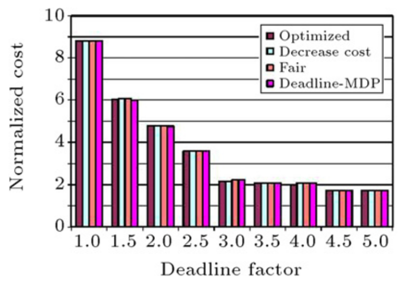
Both algorithms successfully scheduled all workflows before their deadlines, even in the case of tight deadlines, i.e. small deadline factors. Figure 2 shows the cost of scheduling all fifteen workflows (five different workflows, three sizes each) with the SC-PCP (including three scheduling policies) and the Deadline-MDP algorithms, respectively. A quick look at Figure 2 shows that the results for small, medium and large size instances of a particular workflow are almost similar, except for the Large CyberShake and Montage. This is not surprising, because of the infinite resource assumption in the proposed Cloud model. Figure 2 also shows that the optimized policy has the best performance (lowest cost) between three policies for the SC-PCP algorithm in most cases. It also outperforms the Deadline-MDP in many cases. Table 1 shows the average cost decrease percentage of using the SC-PCP algorithm with Optimized policy over the Deadline-MDP algorithm for each workflow. In the next paragraphs, we consider each workflow individually.

CyberShake: This is the only workflow in which both algorithms have almost similar results for small and medium instances. But, Table 1 shows that the SC-PCP with Optimized policy has a better performance, i.e. 21.34% average cost decrease over Deadline-MDP for large instances. The performance of the Decrease Cost policy is as good as the Optimized policy, but the Fair policy behaves similar to the Deadline-MDP.

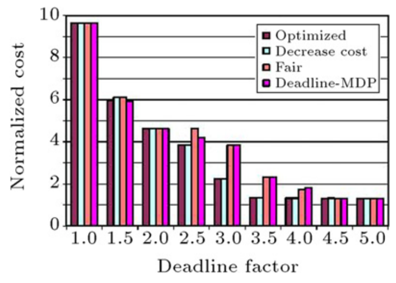
Epigenomics: In this workflow, the SC-PCP has a better performance than the Deadline-MDP in all three instances. Table 1 shows that the performance is better for larger instances of workflow than smaller ones. Except in some cases, the performance of all three policies are almost similar.



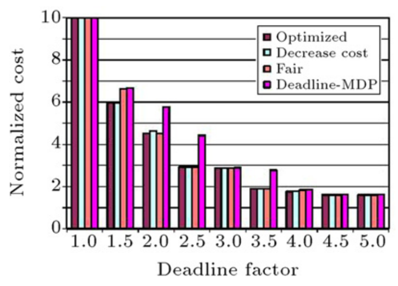
(a) CyberShake (small).



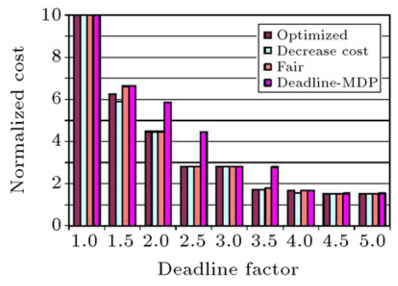
(b) CyberShake (medium).



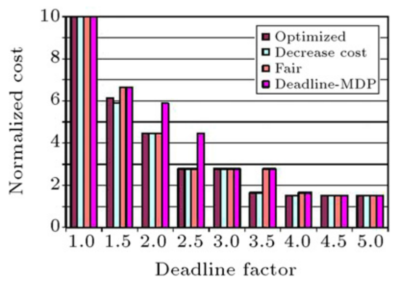
(c) CyberShake (large).



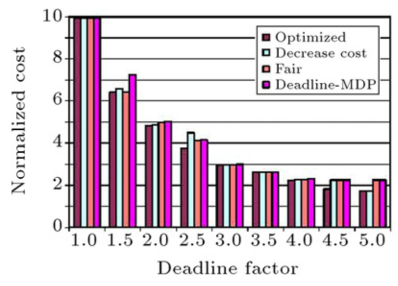
(d) Epigenomics (small).



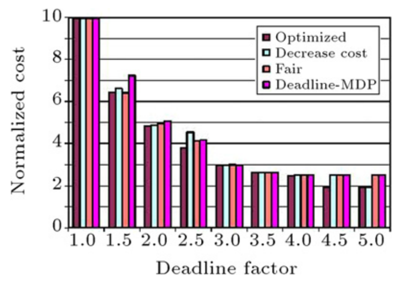
(e) Epigenomics (medium).



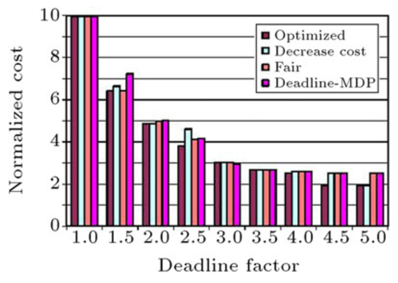
(f) Epigenomics (large).



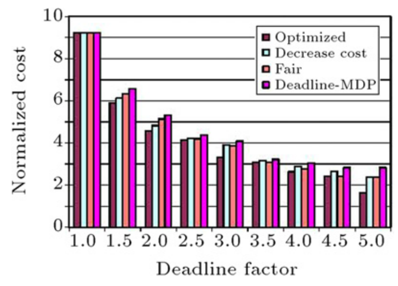
(g) LIGO (small).



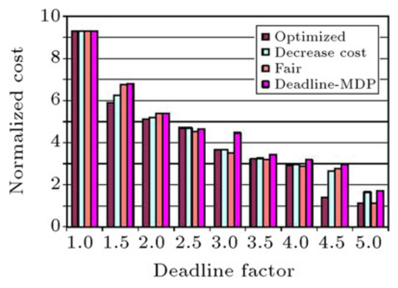
(h) LIGO (medium).



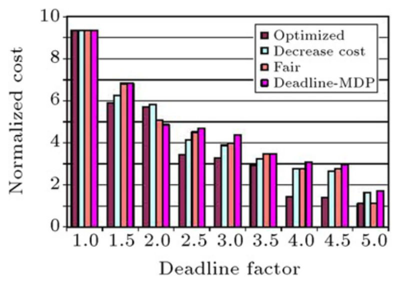
(i) LIGO (large).



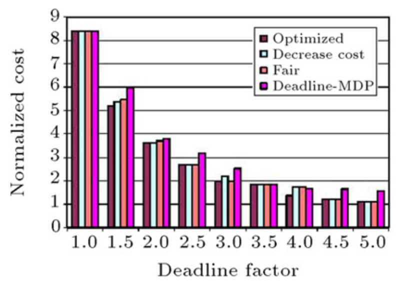
(j) Montage (small).



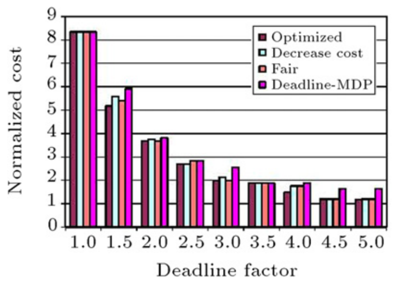
(k) Montage (medium).



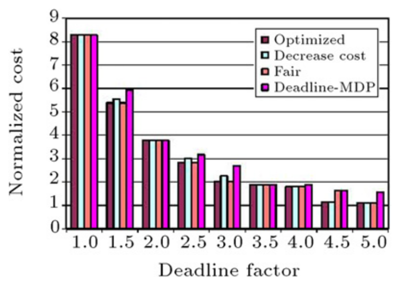
(l) Montage (large).



(m) SIPHT (small).



(n) SIPHT (medium).



(o) SIPHT (large).

Figure 2: Normalized cost of scheduling workflows with the SC-PCP and deadline-MDP algorithms.

Table 2: Maximum computation time of different path scheduling policies for the large size workflows (ms).

	Optimized	Decrease cost	Fair
CyberShake	169	125	122
Epigenomics	35 801	31	27
LIGO	862	24	18
Montage	138	122	127
SIPHT	3 084	60	64

LIGO: Again, the SC-PCP algorithm with Optimized policy has a better performance than Deadline-MDP with approximately 10% average cost decrease. The performances are almost similar in all three sizes.

Montage: This workflow has the best performance for the SC-PCP with Optimized policy, especially in the large instance, with 41.66% average cost decrease over the deadline-MDP. The performance of the algorithm is improved when the workflow gets larger. The performances of the other two policies are not as good as the Optimized policy, but are still better than the Deadline-MDP.

SIPHT: Table 1 shows that the SC-PCP algorithm has a better performance than Deadline-MDP, but the performance decreases when the workflow gets larger. Again, the performances of three policies are almost the same for this workflow.

4.3. Computation time

Table 2 shows the maximum computation time (in milliseconds) of the algorithm for the large size workflows, using three different policies. The computation times of the small and medium size workflows have a similar pattern with smaller values. For the CyberShake workflow, there is no significant difference among the three policies. Remember that the time complexity of the Optimized policy depends on the overall deadline of the workflow. The maximum deadline for this workflow is 930, which is rather small, and has no significant impact on computation time. However, the maximum deadline is 109,355 for the Epigenomics, which results in a very high computation time for the Optimized policy over the other two. The maximum deadline is 4795, 1875 and 21,485 for LIGO, Montage and SIPHT workflows, respectively, which explains their respective computation times.

5. Related work

There are few works addressing workflow scheduling on Clouds. Hoffa et al. [3] 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. [2] compared the performance (in terms of time and cost) of running some scientific workflows on the NCSA's Abe cluster, against the Amazon EC2. Both use Pegasus [6] as the workflow management system to execute the workflows. None of these works consider the cost of running workflows on the Cloud. Salehi and Buyya [22] 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. However, the Cost Optimization only hires extra Cloud resources if it cannot meet the user deadline. However, they do not consider workflow applications. Ostermann et al. [10] consider scheduling workflows on Grid resources, which are capable of leasing Cloud resources, whenever needed. Furthermore, in [23], they extend their work to consider the execution cost of the workflow, in addition to its execution time. However, they use a just-in-time scheduling algorithm, which schedules each task when it becomes ready, unlike our algorithm, which is a complete full-ahead one. Pandey et al. [24] uses Particle Swarm Optimization (PSO) to minimize the execution cost of running workflow applications on the Cloud. Nevertheless, their Cloud model is completely different from ours. Finally, Xu et al. [25] proposed an algorithm for scheduling multiple workflows with multiple QoS constraints on the Cloud.

Moreover, there are some works addressing workflow scheduling on utility Grids, which can be modified to adapt to the Cloud environment. Hereafter, we study some of these works. It is worth noting that there are good heuristic algorithms in this area. However, since heuristic algorithms do not guarantee finding the best solution, there is always room to find better solutions with better heuristics. We have already mentioned the Deadline-MDP proposed by Yu et al. [21]. Sakellariou et al. [26] proposed two scheduling algorithms for a different performance criterion: minimizing the execution time under budget constraints. In the first algorithm, they initially try to schedule workflows with minimum execution time, and then they refine the schedule until its budget constraint is satisfied. In the second one, they initially assign each task to the cheapest resource, and then try to refine the schedule to shorten the execution time under budget constraints.

Prodan et al. [27] 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. First, the DCA algorithm tries to find the best solution, according to the primary criterion, and then tries to optimize this solution for the secondary criterion, while keeping the primary criterion within the predefined sliding constraint.

6. Conclusions

The Cloud computing model enables users to obtain their required services with desired QoS (such as deadline) by paying an appropriate price. In this paper, we propose a new algorithm named the SaaS Cloud Partial Critical Path (SC-PCP) for workflow scheduling in SaaS Clouds, which minimizes the total execution cost while meeting a user-defined deadline. We evaluate our algorithm by simulating it with synthetic workflows that are based on real scientific workflows with different structures and sizes. The results show that SC-PCP outperforms another highly cited algorithm called Deadline-MDP. Furthermore, the experiments show that the computation time of the algorithm is very low for the Decrease Cost and the Fair policies, but is much longer for the Optimized policy, although still acceptable for the mentioned workflows. In the future, we plan to extend our algorithm to support other Cloud computing models, such as IaaS and other pricing models.

References

[1] Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J. and Brandic, I. "Cloud computing and emerging IT platforms: vision, hype and reality for delivering computing as the 5th utility", *Future Gener. Comput. Syst.*, 25(6), pp. 599–616 (2009).

[2] Juve, G., Deelman, E., Vahi, K., Mehta, G., Berriman, B., Berman, B.P. and Maechling, P. "Scientific workflow applications on Amazon EC2", *5th IEEE International Conference on e-Science*, Oxford, UK (2009).

[3] Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B. and Good, J. "On the use of cloud computing for scientific workflows", *Fourth IEEE Int'l Conference on e-Science* (*e-Science 2008*), Indiana, USA, pp. 640–645 (2008).

[4] Deelman, E. "Grids and Clouds: making workflow applications work in heterogeneous distributed environments", *Int. J. High Perform. Comput. Appl.*, 24(3), pp. 284–298 (2010).

[5] Weinhardt, C., Anandasivam, A., Blau, B. and Stoesser, J. "Business models in the service world", *IEEE IT Prof.*, 11(2), pp. 28–33 (2009).

[6] Deelman, E., et al. "Pegasus: a framework for mapping complex scientific workflows onto distributed systems", *Sci. Program.*, 13, pp. 219–237 (2005).

[7] Wieczorek, M., Prodan, R. and Fahringer, T. "Scheduling of scientific workflows in the ASKALON grid environment", *SIGMOD Rec.*, 34(3), pp. 56–62 (2005).

[8] Berman, F., et al. "New grid scheduling and rescheduling methods in the GrADS project", *Int. J. Parallel Program.*, 33(2), pp. 209–229 (2005).

[9] Ramakrishnan, L., et al. "VGrADS: enabling e-science workflows on grids and clouds with fault tolerance", *ACM/IEEE International Conference on High Performance Computing and Communication* (*SC09*), Portland, Oregon, USA (2009).

[10] Ostermann, S., Prodan, R. and Fahringer, T. "Extending grids with cloud resource management for scientific computing", *10th IEEE/ACM International Conference on Grid Computing*, Banff, Alberta, Canada, pp. 42–49 (2009).

[11] Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman (1979).

[12] Kwok, Y.K. and Ahmad, I. "Static scheduling algorithms for allocating directed task graphs to multiprocessors", *ACM Comput. Surv.*, 31(4), pp. 406–471 (1999).

[13] Topcuoglu, H., Hariri, S. and Wu, M. "Performance-effective and low-complexity task scheduling for heterogeneous computing", *IEEE Trans. Parallel Distrib. Syst.*, 13(3), pp. 260–274 (2002).

[14] Bajaj, R. and Agrawal, D.P. "Improving scheduling of tasks in a heterogeneous environment", *IEEE Trans. Parallel Distrib. Syst.*, 15(2), pp. 107–118 (2004).

[15] Wieczorek, M., Hoheisel, A. and Prodan, R. "Towards a general model of the multi-criteria workflow scheduling on the grid", *Future Gener. Comput. Syst.*, 25(3), pp. 237–256 (2009).

[16] Abrishami, S., Naghibzadeh, M. and Epema, D. "Cost-driven scheduling of grid workflows using partial critical paths", *Proceedings of the 11th IEEE/ACM Int'l Conference on Grid Computing* (*Grid2010*), Brussels, Belgium, pp. 81–88 (2010).

[17] Amazon simple storage service (amazon s3). [Online]. Available: http://aws.amazon.com/s3/.

[18] Kellerer, H., Pferschy, U. and Pisinger, D., *Knapsack Problems*, Springer Verlag (2004).

[19] Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.H. and Vahi, K. "Characterization of scientific workflows", *The 3rd Workshop on Workflows in Support of Large Scale Science*, Austin, TX, USA, pp. 1–10 (2008).

[20] Palankar, M.R., Iamnitchi, A., Ripeanu, M. and Garfinkel, S. "Amazon S3 for science grids: a viable solution?" *Proceedings of the 2008 Int'l Workshop on Data-Aware Distributed Computing*, Boston, MA, USA, pp. 55–64 (2008).

[21] Yu, J., Buyya, R. and Tham, C.K. "Cost-based scheduling of scientific workflow applications on utility grids", *First Int'l Conference on e-Science and Grid Computing*, Melbourne, Australia, pp. 140–147 (2005).

[22] Salehi, M.A. and Buyya, R. "Adapting market-oriented scheduling policies for cloud computing", *Proceedings of the 10th Int'l Conference on Algorithms and Architectures for Parallel Processing* (*ICA3PP 2010*), Busan, Korea, pp. 351–362 (2010).

[23] Ostermann, S., Prodan, R. and Fahringer, T. "Dynamic cloud provisioning for scientific grid workflows", *11th IEEE/ACM Int'l Conference on Grid Computing* (*GRID2010*), Brussels, Belgium, pp. 97–104 (2010).

[24] Pandey, S., Wu, L., Guru, S. and Buyya, R. "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments", *24th IEEE Int'l Conference on Advanced Information Networking and Applications* (*AINA*), Perth, Australia, pp. 400–407 (2010).

[25] Xu, M., Cui, L., Wang, H. and Bi, Y. "A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing", *IEEE 11th Int'l Symposium on Parallel and Distributed Processing with Applications*, Chengdu, China, pp. 629–634 (2009).

[26] Sakellariou, R., Zhao, H., Tsiakkouri, E. and Dikaiakos, M.D. "Scheduling workflows with budget constraints", In *Integrated Research in GRID Computing*, S. Gorlatch and M. Danelutto, Eds., pp. 189–202, Springer-Verlag (2007).

[27] Prodan, R. and Wieczorek, M. "Bi-criteria scheduling of scientific grid workflows", *IEEE Trans. Autom. Sci. Eng.*, 7(2), pp. 364–376 (2010).

Saeid Abrishami received his B.S. and M.S. degrees in Computer Engineering from Ferdowsi University of Mashhad, in 1997 and 1999, respectively. In 2006, he started his Ph.D. degree in Computer Engineering at the same university. During the spring and summer of 2009, he was visiting researcher at Delft University of Technology, in Holland. His research interests focus on resource management and scheduling in distributed systems, especially utility grids and Clouds.

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