Cost-driven Scheduling of Grid Workflows Using Partial Critical Paths

Saeid Abrishami, Mahmoud Naghibzadeh
Ferdowsi University of Mashhad
Mashhad, Iran
{s-abrishami, naghibzadeh}@um.ac.ir

Dick Epema
Delft University of Technology
Delft, The Netherlands
d.h.j.epema@tudelft.nl

***Abstract*—Recently, utility grids have emerged as a new model of service provisioning in heterogeneous distributed systems. In this model, users negotiate with providers on their required Quality of Service and on the corresponding price to reach a Service Level Agreement. One of the most challenging problems in utility grids is workflow scheduling, i.e., the problem of satisfying users' QoS 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 Path. This algorithm recursively schedules the critical path ending at a recently scheduled node. The proposed algorithm tries to minimize the cost of workflow execution while meeting a user-defined deadline. The simulation results show that the performance of our algorithm is very promising.**

***Index Terms*—grid computing, workflow scheduling, utility grids, economic grids, QoS-based scheduling**

I. Introduction

Many researchers believe that economic principles will influence the grid computing paradigm to become an open market of distributed services, sold at different prices, with different performance and QoS [1]. This new paradigm is known as *utility grid*, versus the traditional community grid in which services are provided free of charge with best-effort service. Although there are many papers that address the problem of scheduling in traditional grids, there are only a few works on this problem in utility grids. The multi-objective nature of the scheduling problem in utility grids makes it difficult to solve, specially 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 utility grids, and we evaluate its performance on some well-known scientific workflows in the grid context.

The main difference between community grids and utility grids is QoS: while community grids follow the best-effort method in providing services, utility grids guarantee the required QoS of users via Service Level Agreements (SLAs) [2]. An SLA is a contract between the provider of resources and the consumer of those resources describing the qualities and the guarantees of the service provisioning. Consumers can negotiate with providers on required QoS and the price to reach an SLA. The price has a key role in this contract: it encourages providers to advertise their services to the market, and encourages consumers to define their required qualities more realistically. Obviously, traditional resource management systems for community grids are not directly suitable for utility grids, and therefore, new methods have been proposed and implemented in recent years [3].

Workflows constitute a common model for describing a wide range of applications in distributed systems. 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. 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 [4], many heuristic methods have been proposed for homogeneous [5] and heterogeneous distributed systems like grids [6], [7], [8], [9], [10], [11]. These scheduling methods try to minimize the execution time (makespan) of the workflows and as such are suitable for community grids. However, in utility grids, there are many potential other QoS attributes other than execution time, like reliability, security, availability, and so on. Besides, higher QoS attributes mean higher prices for the services. Therefore, the scheduler faces a QoS-cost tradeoff in selecting appropriate services.

In this paper we propose a new QoS-based workflow scheduling algorithm, called the *Partial Critical Paths* (PCP) algorithm. The objective function of the 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 PCP algorithm tries to schedule the (overall) critical path of the workflow such that it completes before the user's deadline and the 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 II describes our system model, including the application model, the utility grid model, and the objective function. The PCP scheduling algorithm is explained in Section III. A performance evaluation is presented in Section IV, and Section VI concludes.

978-1-4244-9349-4/10/$26.00 © 2010 IEEE

II. Scheduling System Model

The proposed scheduling system model consists of an application model, a utility grid 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 $\{t_1, t_2, ..., t_n\}$, and E is a set of arcs. Each arc $e_{i,j} = (t_i, t_j)$ represents a precedence constraint that 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 arcs to the actual entry and exit tasks.

A utility grid model consist of several Grid Service Providers (GSPs), each of which provides some services to the users. Each workflow task t_i can be processed by m_i services $S_i = \{s_i^1, s_i^2, ..., s_i^{m_i}\}$ from different service providers with different QoS attributes. There are many QoS attributes for services, like execution time, price, reliability, security, and so on. In this study we use the most important ones, execution time and price, for our scheduling model. The price of a service usually depends on its execution time, i.e., shorter execution times are more expensive. However, some service providers may offer special services to special users, or in certain (off-peak) times. We assume $ET(t_i, s)$ and $EC(t_i, s)$ to be the estimated execution time and execution cost for processing task t_i on service s, respectively. Estimating the execution time of a task on an arbitrary resource is an important issue in grid scheduling. Many techniques have been proposed in this area such as code analysis, analytical benchmarking/code profiling, and statistical prediction [12], that are beyond our discussion. Besides, there is another source of time and money consumption: transferring data between tasks. We assume $TT(e_{i,j}, s, r)$ and $TC(e_{i,j}, s, r)$ to be the estimated transfer time and transfer cost of sending the required data along $e_{i,j}$ from service *s* (processing task t_i) to service *r* (processing task t_j), respectively. Estimating the transfer time can be done using the amount of data to be transmitted, and the bandwidth and latency information between services.

To obtain the available services and their information, the scheduler should query a grid information service like the Grid Market Directory (GMD)[13]. In a utility grid, GMD is used to provide information such as the type, the provider, and the QoS parameters (including price) for all services. Each GSP has to register itself and its services with the GMD, so that it can present and sell its services to users. Whenever a scheduler accepts a workflow, it contacts the GMD to query about available services for each task and their QoS attributes. Then the broker directly contacts the service's GSP to gather detailed information about the dynamic status of the service, especially the available time slots for processing tasks. Using this information, the scheduler can execute a scheduling algorithm to map each task of a workflow to one of the available services. According to the generated schedule, the broker contacts GSPs to make advance reservations of selected services. This results in an SLA between the broker and the GSP specifying the earliest start time, the latest finish time, and the price of the selected service. Usually the SLA contains a penalty clause in case of violation of the service level to enforce service level guarantees.

The last parameter in our model is the performance criteria. In community grids (traditional scheduling), users prefer to minimize the completion time (makespan) of their jobs. However, in utility grids, price is the most important factor. Therefore, users prefer to utilize cheaper services with lower QoS that satisfy their needs and expectations. 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 criteria are to minimize the execution cost of the workflow such that the workflow is complete before the user's specified deadline.

III. The Proposed Algorithm

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* (*node*), 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 that 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 our *Partial Critical Path* (PCP) algorithm, this procedure continues recursively until all tasks are scheduled successfully. In the following sections, we elaborate on the details of the PCP algorithm.

A. Basic Definitions

In our PCP scheduling algorithm, we want to find the critical path of the whole workflow, and partial critical paths. In order to find these, we need some (idealized, approximate) notion of the start time of each workflow task before we actually schedule the task. This means that we have two notions of the start times of tasks, the earliest start time computed before scheduling the workflow, and the actual start time computed by our scheduling algorithm.

For each unscheduled task t_i we define its Earliest Start Time $EST(t_i)$ as the earliest time t_i can start its computation, regardless of the actual service that will process the task (that will be determined during scheduling). Clearly, it is not possible to compute $EST(t_i)$ exactly, because a grid is

Algorithm 1 The PCP Scheduling Slgorithm

```
 1: procedure SCHEDULEWORKFLOW(G(T, E), deadline)
 2:     request available services for each task in G from GMD
 3:     query available time slots for each service from related GSPs
 4:     add t_entry, t_exit and their corresponding edges to G
 5:     compute MET(t_i) for each task according to formula 1
 6:     compute MTT(e_i,j) for each edge according to formula 2
 7:     compute EST(t_i) for each task in G according to formula 3
 8:     mark t_entry and t_exit as scheduled
 9:     set AST(t_entry) ← 0, AST(t_exit) ← deadline
10:     if (ScheduleParents(t_exit) is successfull) then
11:         make advance reservations for all tasks in G according to Schedule
12:     else
13:         return (failure)
14:     end if
15: end procedure
```

a heterogeneous environment and the computation time of tasks varies from service to service. Furthermore, the data transmission time is also dependent on the selected services and the bandwidth between their providers. Thus, we have to approximate the execution and data transmission time for each unscheduled task. Among the possible approximation options, e.g., the average, the median, or the minimum, we select the minimum execution and data transmission time. We define the Minimum Execution Time $MET\,(t_i)$ and the Minimum Transmission Time $MTT\,(e_{i,j})$ as follows:

$$MET\,(t_i) = \min_{s \in S_i} ET\,(t_i, s) \tag{1}$$

$$MTT\,(e_{i,j}) = \min_{s \in S_i, r \in S_j} TT\,(e_{i,j}, s, r) \tag{2}$$

Having these definitions, we can compute the Earliest Start Time for each unscheduled task, $EST\,(t_i)$, as follows:

$$\begin{aligned} EST\,(t_{entry}) &= 0 \\ EST\,(t_i) &= \max_{t_p \in parents\ t_i} EST\,(t_p) + MET\,(t_p) + MTT\,(e_{p,i}) \end{aligned} \tag{3}$$

For each scheduled task we define the Selected Service $SS\,(t_i)$ as the service selected for processing t_i during scheduling, and the Actual Start Time $AST\,(t_i)$ as the actual start time of t_i on that service. These attributes will be determined during scheduling.

B. The PCP Scheduling Algorithm

Algorithm 1 shows the pseudo-code of the overall PCP algorithm for scheduling a workflow. In line 4, two dummy nodes t_{entry} and t_{exit} have been added to the task graph, even if the task graph already has only one entry or exit node. This is necessary for our algorithm, but we won't actually schedule these two tasks. After computing the required parameters in lines 5 - 7, dummy nodes t_{entry} and t_{exit} are marked as scheduled in line 8, and then the Actual Start Time of t_{exit} is set to the user's deadline. This enforces the parents of t_{exit}, i.e., the actual exit nodes of the workflow, to be finished before the deadline. Finally, in line 10 the function ScheduleParents is called for t_{exit}. In order to show how the algorithm works, we will trace its operation on the sample graph shown in Figure 1 (the numbers simply indicate node numbers, not execution times). In this figure, t_{entry} and t_{exit} have been shown with *S* and *E*, respectively. In this stage, ScheduleWorkflow calls ScheduleParents for node *E*.

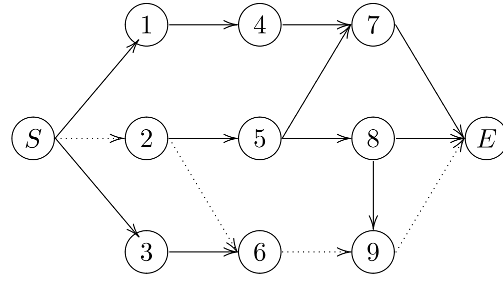


Figure 1. A sample workflow

C. The Parents Scheduling Algorithm

The pseudo-code for ScheduleParents is shown in algorithm 2. This algorithm receives a scheduled node as input and tries to schedule all of its parents before the actual start time of the input node itself. On success, it returns the desired schedule, but on failure, it returns a task that causes this failure and a suggested start time for this task that hopefully makes its scheduling possible. First, ScheduleParents tries to find the *Partial Critical Path* of unscheduled nodes ending at the 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 a node t is the unscheduled parent of t that has the latest data arrival time at t, that is, it is the parent p of t for which $EST(p) + MET(p) + MTT(e_{p,t})$ is maximal.*

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

Definition 2. *The Partial Critical Path of node t is:*

i *empty if t does not have unscheduled parents.*
ii *consists of the Critical Parent p of t and the Partial Critical Path of p if t has 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 5-10). 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. In the example workflow of Figure 1, the real critical path is shown with dotted arrows. Therefore, in this stage CriticalPath will be set to 2-6-9.

Then the algorithm calls procedure SchedulePath, which receives a path (an ordered list of nodes) and a list of start time constraints for all nodes on the path as input. If SchedulePath succeeds, it returns a schedule for this path with the minimum price that satisfies the constraints list and ends before the start time of the scheduled children of the last node in the path. If it fails, it returns a task that causes this failure and a suggested start time for it. Initially, all of the constraints of the nodes

Algorithm 2 Parents Scheduling Algorithm

```
1: procedure SCHEDULEPARENTS(t)
2:     if (t has no unscheduled parent) then
3:         return (Success)
4:     end if
5:     t_i ← t
6:     CriticalPath ← empty
7:     while (there exists an unscheduled parent of t_i) do
8:         add CriticalParent(t_i) to the beginning of CriticalPath
9:         t_i ← CriticalParent(t_i)
10:    end while
11:    initialize Constraints to 0
12:    while (CriticalPath is not scheduled) do
13:        if (SchedulePath(CriticalPath, Constraints) is unsuccessful) then
14:            set t_failure and SuggestedStartTime and return (Failure)
15:        end if
16:        for all (t_i ∈ CriticalPath) do
17:            if (ScheduleParents(t_i) is unsuccessful) then
18:                if (t_failure ∈ CriticalPath) then
19:                    Constraints[t_failure] ← SuggestedStartTime
20:                    break out from for loop
21:                else
22:                    set t_failure and SuggestedStartTime and return (Failure)
23:                end if
24:            end if
25:        end for
26:    end while
27:    return ScheduleParents(t)
28: end procedure
```

have been set to zero (line 11). If SchedulePath fails (lines 13-15), it will do so because of one of the nodes that have been scheduled previously in higher levels (previous calls of this algorithm). In other words, SchedulePath cannot schedule a particular task of the path, because one of the the previously scheduled children of this task has a start time that is to early to be met. In this case ScheduleParents returns the previously scheduled task that causes this failure with a suggested start time for it.

If SchedulePath succeeds (lines 16-25), 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. If one of these recursive calls to ScheduleParents fails, there are two different situations: either the task that causes this failure belongs to the current partial critical path, or it does not[1] (lines 17-24). In the former case, we cancel all schedules until now and call SchedulePath to schedule the partial critical path again, but with a new constraint for the failed task. In the latter case, we can't do anything in this stage, so we return failure and the failed task to the caller algorithm to reschedule it. This process continues until all tasks on the partial critical path and their parents have successfully been scheduled. At the end, ScheduledParents is called once again for the input node (line 27) to schedule its remaining unscheduled parents (if any exists).

Let's now follow the algorithm on the example workflow of Figure 1. After finding the critical path 2-6-9, it calls SchedulePath to find the best schedule for this path. Then, ScheduleParents will be called for each node on this path to find its partial critical path:

- Node 2: has no unscheduled parent
- Node 6: its partial critical path consists of only node 3, so SchedulePath is called to schedule node 3. If SchedulePath cannot schedule node 3, it returns failure and the node responsible for this failure, which is node 6, and a suggested new start time for this node. Then SchedulePath should schedule path 2-6-9 again with this new constraint for node 6. But if SchedulePath succeeds, the algorithm continues to the next node.
- Node 9: its partial critical path is 5-8, which will be scheduled by calling SchedulePath. Again, if SchedulePath cannot schedule this path, it returns failure, forcing path 2-6-9 to be rescheduled.

[1]Remember the task causes this failure belongs to the previously scheduled tasks, so it can be on the current partial critical path that has been scheduled recently

Algorithm 3 Path Scheduling Algorithm

```
1: procedure SCHEDULEPATH(Path, Constraints)
2:     bestSchedule ← null
3:     t ← first task on the path
4:     while (t is not null) do
5:         s ← next untried service ∈ S_t
6:         if (s = Ø) then
7:             t ← previous task on the path and continue while loop
8:         end if
9:         Compute ST(t, s) and C(t, s)
10:        if (ST(t, s) < Constraints_t) then
11:            ST(t, s) ← Constraints_t
12:        end if
13:        for all (nodes sc ∈ Scheduled Children of t) do
14:            if (Actual Start Time of sc can not be met) then
15:                continue while loop
16:            end if
17:        end for
18:        if (t is the last task on the Path) then
19:            if (this schedule has a better cost than bestSchedule) then
20:                set this schedule as the bestSchedule
21:                t ← previous task on the path
22:            end if
23:        else
24:            t ← next task on the path
25:        end if
26:    end while
27:    if (an admissible schedule found) then
28:        mark all nodes of Path as scheduled
29:        set AST(t) ← ST(t, bestSchedule_t) for all tasks t in Path
30:        update EST for all unscheduled children of all tasks in Path
31:        return (Success)
32:    else
33:        determine t_failure and a suggested start time for it
34:        return (Failure)
35:    end if
36: end procedure
```

Now ScheduleParents finishes the scheduling of the partial critical path for node *E*. Then it calls itself again to schedule the unscheduled parents of node *E* (line 27). There is only one unscheduled parent remaining for E, which is node 7, and finding its partial critical path results in 1-4-7, that will be scheduled by calling SchedulePath. If the scheduling is successful, the algorithm finishes. Otherwise, it returns failure, causing the previous stages to be rescheduled with new constraints.

D. The Path Scheduling Algorithm

The last algorithm, SchedulePath, which is shown in algorithm 3, is based on a Backtracking strategy. It starts from the first task in the path and moves forward to the last task, at

each step selecting an untried available service for that task (line 5). If the selected service creates an admissible (partial) schedule, then it moves forward to the next task, otherwise it selects another untried service for that task. If there is no available untried service for that task left, then it backtracks to the previous task on the path and selects another service for it (lines 6-8). After selecting a service for the current task *t*, say service *s*, the algorithm computes the start time *ST(t,s)* and the actual cost *C(t,s)* of running task *t* on service *s*; *ST(t,s)* is computed by considering the data arrival time of the *scheduled* parents of *t* and the free time slots of *s*. The algorithm finds a free slot of length at least equal to the execution time of *t* on *s* that begins after receiving all the required data from the scheduled parents of *t* at *s*. The actual cost *C(t,s)* is computed by summing three parts:

- the execution cost of t on s
- the cost of the data transmission between selected services for *scheduled* parents of t (including its parent on the path) and s
- the cost of the data transmission between s and selected services for *scheduled* children of t

Then the start time constraint for current task is checked (line 10). After that, the algorithm checks the admissibility of the current (partial) schedule (lines 13-17) by checking whether the actual start times of the scheduled children of the current task can be met. Finally, if the scheduling was successful, the AST for the scheduled task and the EST for the children of them are updated and the algorithm returns success with the corresponding schedule (lines 27-31). Otherwise, the task that causes the failure and its Suggested Start Time are returned (lines 32-34), the details has been omitted for the sake of brevity.

IV. Performance Evaluation

In this section we will present our simulations of the Partial Critical Path algorithm.

A. Experimental Setup

We have used GridSim [14] for simulating the utility grid environment for our experiments. We simulate a grid environment like DAS-3[15] which is a multicluster grid in the Netherlands. It consists of five clusters, and comprises 272 dual-processor AMD Opteron compute nodes. The constitutive clusters are: Vrije University with 85 nodes, University of Amsterdam with 41 nodes, Delft University with 68 nodes, MultimediaN with 46 nodes and Leiden University with 32 nodes. The average inter-cluster bandwidth is between 10 to 512 MB/s. Unfortunately the processor speeds are very close (between 2.2 to 2.6 GHz), so we have changed them to make a 10 times difference between the fastest (Leiden University) and the slowest (U. of Amsterdam) cluster. Besides, as DAS-3 is not a utility grid, we assigned fictitious prices (between 0.5 to 50 G$ per second) to each cluster that follow the rule that a faster cluster has a higher price.

We test our algorithm using five synthetic workflow applications that are described in [16]. These workflows are based on real scientific workflows: Montage, CyberShake, Epigenomics, LIGO and SIPHT. These applications have different structural properties in terms of their basic components (pipeline, data aggregation, data distribution and data redistribution) and their composition. Furthermore, there are four different sizes for each workflow application in terms of total number of tasks, from which we select three sizes: small (about 30 tasks), medium (about 100 tasks) and large (about 1000 tasks). We assume that all services are installed on all clusters, such that each task can be executed on every cluster. In addition, we assume that all clusers are empty in the beginning.

B. Experimental Results

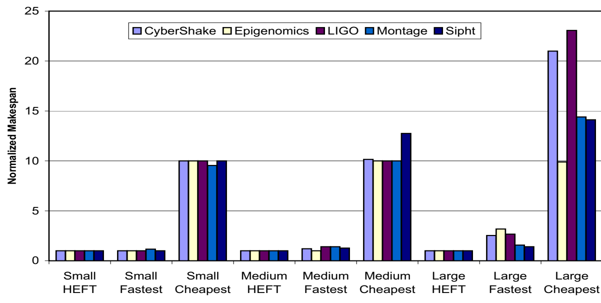
First, to get a better idea of the required time and cost for each workflow application, we simulate their execution using three scheduling algorithms: *HEFT* [6], a well-known makespan minimization algorithm, *Fastest,* that submits all tasks to the fastest cluster, and *Cheapest,* that submits all tasks to the cheapest (and slowest) cluster. Note that the last two algorithms submit all tasks to one cluster (fastest or cheapest), therefore some tasks have to wait for free resources, particularly in the case of large workflows. Furthermore, since a large set of workflows with different attributes is used, it is important to normalize the total cost and makespan of each workflow execution. So we define the Normalized Cost (NC) and the Normalized Makespan (NM) of a workflow execution as follows:

$$NC = \frac{total\ schedule\ cost}{C_C} \quad (4)$$

$$NM = \frac{schedule\ makespan}{M_H} \quad (5)$$

where C_C is the cost of executing the same workflow with the *Cheapest* strategy and M_H is the makespan of executing the same workflow with the *HEFT* strategy. The results of executing workflow applications using these three scheduling policies are shown in Figure 2. Obviously, the normalized cost and the normalized makespan are the same for the HEFT and Fastest strategy for small workflows, because the number of tasks is less than the fastest cluster's resources. But they are slightly different in medium workflows, and in large workflows they have a meaningful difference because the HEFT strategy tries to send some tasks to slower resources rather than waiting for the fastest resource to finish the currently assigned tasks. This strategy decreases the makespan and cost at the same time. The price of the fastest resource is 10 times more than the slowest resource, so the maximum normalized cost is 10.

To evaluate our 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 HEFT 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 $\alpha \cdot M_H$. In our experiments, we let α range from 1.5 to 5.



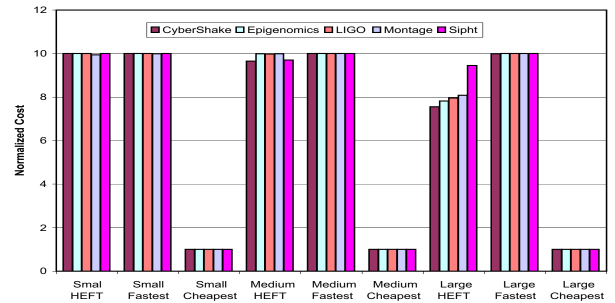
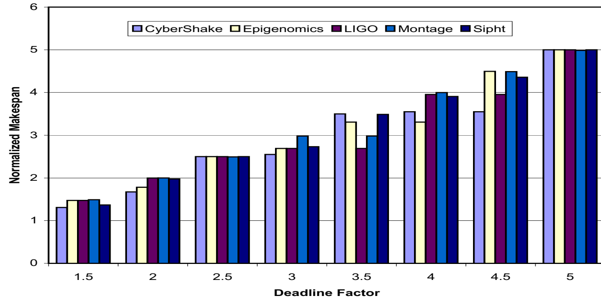


Figure 2. Normalized Makespan (left) and Normalized Cost (right) of scheduling workflows with three scheduling policies: HEFT, Fastest and Cheapest



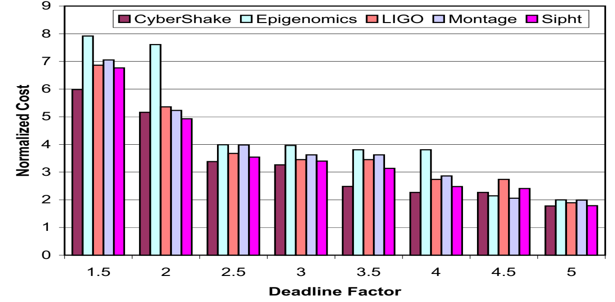
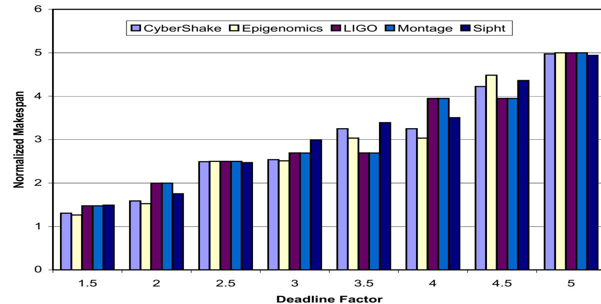


Figure 3. Normalized Makespan (left) and Normalized Cost (right) of scheduling of **small** workflows with Partial Critical Paths algorithm



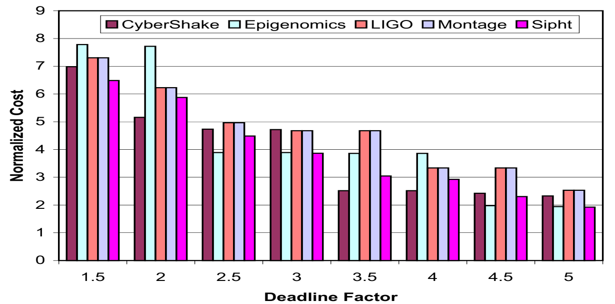
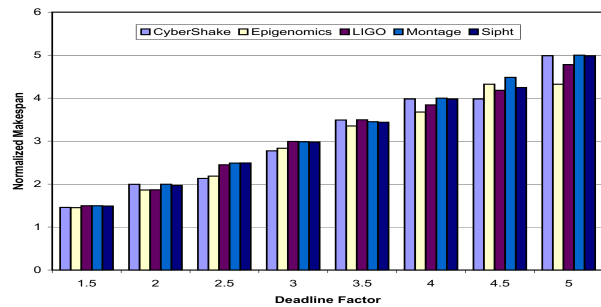


Figure 4. Normalized Makespan (left) and Normalized Cost (right) of scheduling of **medium** workflows with Partial Critical Paths algorithm



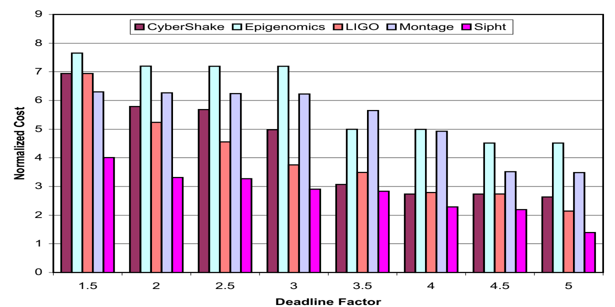
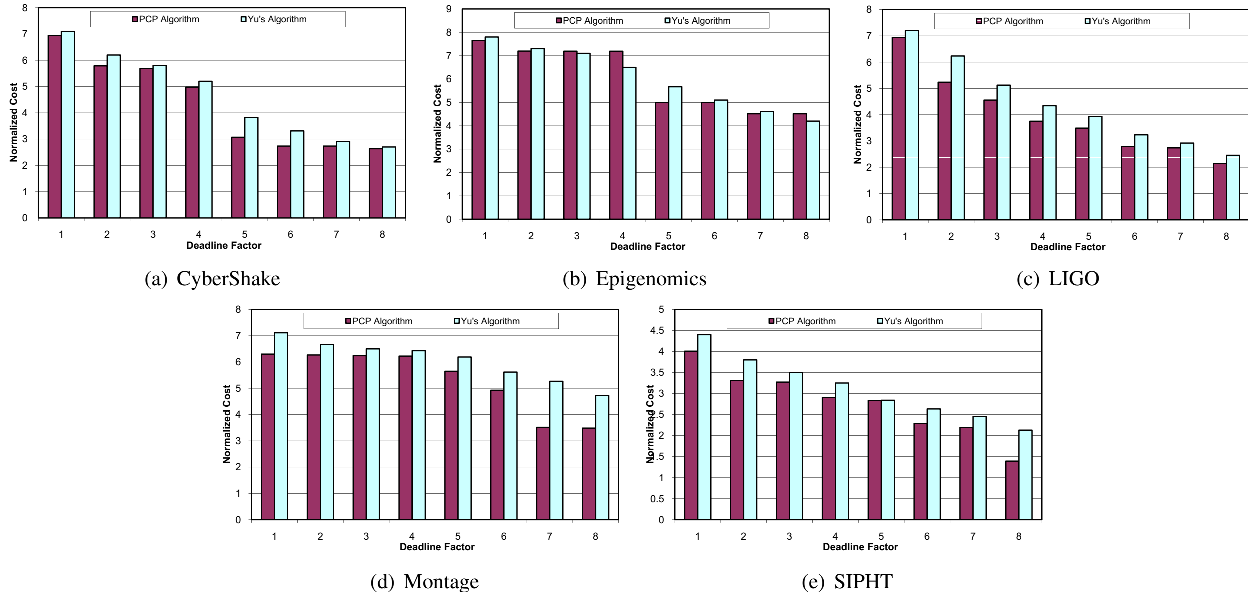


Figure 5. Normalized Makespan (left) and Normalized Cost (right) of scheduling of **large** workflows with Partial Critical Paths algorithm

The normalized makespan and the normalized cost are shown in Figure IV-A for small workflows. As can be seen, the PCP algorithm meets the deadline in all cases and the normalized cost decreases with the deadline increase. However, in some cases, increasing the deadline does not change the schedule, and the cost remains the same, e.g., when going from $\alpha = 3$ to $\alpha = 3.5$ for the LIGO and Montage workflows. In these cases, the deadline increase is not sufficient to assign tasks to slower resources, so the algorithm prefers to keep the previous schedule. Although some workflows have a better performance improvement in the first step, e.g., the normalized cost of CyberShake decreases from 10 to 6 when the deadline



(a) CyberShake (b) Epigenomics (c) LIGO

(d) Montage (e) SIPHT

Figure 6. Normalized Cost of scheduling of the large workflows with PCP and Yu's algorithm

increases from M_H to $1.5M_H$, in the end all the workflows have an almost similar normalized cost. This means that when we increase the deadline about 5 times from M_H to $5M_H$, the normalized cost decreases to slightly less than twice C_C, which is a promising performance.

Figures 4 and 5 show the normalized makespan and the normalized cost for the medium and large workflows, respectively. For the medium workflows, the charts are more or less similar those for the small ones, but for the large workflows things are completely different for some workflows. The Epigenomics workflow has the worst performance with a decrease in the normalized cost only to 4.5, while the SIPHT workflow has the best performance with a decrease of the normalized cost to 1.39 (even better than its small and medium versions) for $\alpha = 5$. This shows that in large workflows with huge numbers of tasks (about 1000), the structural properties of the workflows influence the scheduling process more than for small and medium ones.

In some workflows, the normalized cost decreases smoothly, while in the others it remains constant for a while and then decreases suddenly. This is because of the structural properties of each workflow. For example, in the Epigenomics workflow a sudden decrease in the normalized cost is visible. This workflow consists of many parallel pipelines with four tasks each. The first three tasks have very small runtimes (less than a second) but the fourth is very time consuming (about 942 seconds on the fastest machine on average). So, when this workflow executes, there are many instances of the fourth task that are executing at the same time. When we have a tight deadline, all of these instances are submitted to the fast resources. Sending even one of these tasks to a slow (and also cheap) resource, drastically increases the makespan of the workflow. So, small increases in the deadline don't change the schedule until the deadline reaches a limit that can send all (or most) of the instances of the fourth task to the slower (and cheaper) resources. At that point, the total cost of the schedule suddenly reduces.

In summary, we can conclude that the PCP algorithm has a promising performance in decreasing the normalized cost of the workflows when the deadline increases for small and medium workflows (less than or equal to 100 tasks), but that in large workflows (about 1000 tasks) its performance depends on the structural properties of the workflows.

Some readers may concern about the runtime of our algorithm because if the algorithm fails to schedule a path, it has to reschedule some previously scheduled paths. Fortunately rescheduling occures rarely. We run the PCP algorithm on a two cores 2.1 GHz Intel CPU. The runtime is less than a second for the small and medium workflows. For the large workflows, it takes about 5 to 15 seconds, depending on the workflow and the deadline.

C. Comparing to Other Algorithms

In this section, we have compared the PCP algorithm with one of the most cited algorithms in this area, that has been proposed by Yu et al. [17]. 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 try to minimize cost of each partition execution under sub-deadline constraint. We repeat the same experiments of section IV-B for this algorithm. Figure 6 compare the Normalized Cost that has been generated by each algorithm for the large workflows. It can be seen that in most cases our algorithm has a better performance (lower cost) than Yu's algorithm. Table I shows the minimum, maximum and average of the percentage of the cost decrease for each workflow. The Epigenomics workflow is the only one that has a negative improvement for our algorithm, this

Table I
RANGE OF COST DECREASE PERCENTAGE OF THE PCP OVER THE YU'S ALGORITHM

Workflow	Min	Max	Average
CyberShake	2.01	19.63	7.57
Epigenomics	-10.64	11.90	-0.0048
LIGO	3.61	15.99	11.04
Montage	3.20	33.26	13.14
SIPHT	0.28	34.60	12.19

is because of the problem that we have discussed in section IV-B. As both algorithms manage to end before the specified deadline, we omit the makespan charts. The results are mostly similar for the small and medium workflows.

V. RELATED WORK

There are few works addressing workflow scheduling with QoS in the literature, most of them consider the execution time of the workflow as the major QoS attribute. We have already mentinoned the algorithm proposed by Yu et al. [17] for minimizing the cost of workflow execution under deadline constraints. Sakellariou et al. [18] proposed two scheduling algorithm 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 satisfied. In the second algorithm, they initially assign each task to the cheapest resource, and then try to refine the schedule to shorten the execution time under budget constraints. Other methods like Integer Programming [19], Mixed-Integer Non-Linear Programming [20], and Game Theory [21] are also used for this problem. In addition, some researchers use Metahuristics like Genetic Algorithm [2], Ant Colony Optimization [22], Tabu Search, Simulated Annealing and Guided Local Search [23], and Multiobjective differential evolution [24] to solve this problem.

VI. CONCLUSIONS

Utility grids enable users to obtain their desired QoS (such as deadline) by paying an appropriate price. In this paper we propose a new algorithm for workflow scheduling in utility grids that 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 the results show that it has a promising performance in small and medium workflows, but that its performance in large workflows is variable and depends on the structure of the workflow. In the future, we will extend our algorithm to support other economic grid models and also try to enhance it for the cloud computing model.

REFERENCES

[1] D. Laforenza, "European strategies towards next generation grids," in *Proc. of The Fifth Int'l Symposium on Parallel and Distributed Computing (ISPDC '06)*, 2006, p. 11.

[2] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program.*, vol. 14, pp. 217–230, 2006.

[3] J. Broberg, S. Venugopal, and R. Buyya, "Market-oriented grids and utility computing: The state-of-the-art and future directions," *J. Grid Comput.*, vol. 6, pp. 255–276, 2008.

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

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

[6] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 2002.

[7] R. Bajaj and D. P. Agrawal, "Improving scheduling of tasks in a heterogeneous environment," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, pp. 107–118, 2004.

[8] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *Proc. of the Third IEEE Int'l Conference on e-Science and Grid Computing (E-SCIENCE '07)*, 2007, pp. 35–42.

[9] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *J. of Parallel and Distributed Computing*, vol. 68, pp. 399–409, 2008.

[10] R. Sakellariou and H. Zhao, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *Proc. of the 18th Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004, pp. 111–.

[11] D. Bozdag, U. Catalyurek, and 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, pp. 12–.

[12] S. Verboven, P. Hellinckx, F. Arickx, and J. Broeckhove, "Runtime prediction based grid scheduling of parameter sweep jobs," in *Asia-Pacific Conference on Services Computing*, 2008, pp. 33–38.

[13] J. Yu, S. Venugopal, and R. Buyya, "A market-oriented grid directory service for publication and discovery of grid service providers and their services," *The Journal of Supercomputing*, vol. 36, pp. 17–31, 2006.

[14] R. Buyya and M. Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurrency and Computation: Practice and Experience(CCPE)*, vol. 14, pp. 1175–1220, 2002.

[15] The distributed asci supercomputer. [Online]. Available: http://www.cs.vu.nl/das3/

[16] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *The 3rd Workshop on Workflows in Support of Large Scale Science*, 2008.

[17] J. Yu, R. Buyya, and 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.

[18] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling workflows with budget constraints," in *Integrated Research in GRID Computing (CoreGRID Integration Workshop 2005, Selected Papers)*, S. Gorlatch and M. Danelutto, Eds., 2007.

[19] I. Brandic, S. Benkner, G. Engelbrecht, and R. Schmidt, "QoS support for time-critical grid workflow applications," in *Int'l Conference on e-Science and Grid Computing*, 2005, pp. 108–115.

[20] A. Afzal, J. Darlington, and A. McGough, "QoS-Constrained stochastic workflow scheduling in enterprise and scientific grids," in *Proc. of the 7th IEEE/ACM Int'l Conference on Grid Computing (GRID '06)*, 2006, pp. 1–8.

[21] R. Duan, R. Prodan, and T. Fahringer, "Performance and cost optimization for multiple large-scale grid workflow applications," in *Proc. of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–12.

[22] W.-N. Chen and J. Zhang, "An ant colony optimization approach to grid workflow scheduling problem with various QoS requirements," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 39, pp. 29–43, 2009.

[23] D. M. Quan and D. F. Hsu, "Mapping heavy communication grid-based workflows onto grid resources within an SLA context using metaheuristics," *Int'l J. High Perform. Comput. Appl.*, vol. 22, pp. 330–346, 2008.

[24] A. K. M. K. A. Talukder, M. Kirley, and R. Buyya, "Multiobjective differential evolution for workflow execution on grids," in *Proc. of the 5th Int'l workshop on Middleware for grid computing (MGC '07)*, 2007, pp. 1–6.