Delft University of Technology
Parallel and Distributed Systems Report Series

# Cost-driven Scheduling of Grid Workflows Using Partial Critical Paths

Saeid Abrishami, Mahmoud Naghibzadeh, Dick Epema
s-abrishami@um.ac.ir,naghibzadeh@um.ac.ir, D.H.J.Epema@tudelft.nl

Completed April 2011. Submitted for publication

report number PDS-2011-001

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://www.pds.ewi.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 service 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 the QoS of the users 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), that tries to minimize the cost of workflow execution while meeting a user-defined deadline. The PCP algorithm has two phases: in the deadline distribution phase it recursively assigns sub-deadlines to the tasks on the partial critical paths ending at previously assigned tasks, and in the planning phase it assigns the cheapest service to each task while meeting its sub-deadline. The simulation results show that the performance of the PCP algorithm is very promising.

# Contents

# List of Figures



# List of Tables

# 1 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 Quality of Service (QoS) [2]. 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) [3]. 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 the 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 [4].

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. Due to the importance of workflow applications, many Grid projects such as Pegasus [5], ASKALON [6], and GrADS [7] have designed workflow management systems to define, manage, and execute workflows on the Grid. A taxonomy of Grid workflow management systems can be found in [8]. 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 [9], many heuristic methods have been proposed for homogeneous [10] and heterogeneous distributed systems like Grids [11, 12, 13, 14]. These scheduling methods try to minimize the execution time (makespan) of the workflows and as such are suitable for community Grids. Most of the current workflow management systems, like the ones above mentioned, use such scheduling methods. However, in utility Grids, there are many potential other QoS attributes than execution time, like reliability, security, availability, and so on. Besides, stricter QoS attributes mean higher prices for the services. Therefore, the scheduler faces a QoS-cost tradeoff in selecting appropriate services, which belongs to the *multi-objective optimization problems* family.

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. The proposed algorithm has two main phases: Deadline Distribution and Planning. In the former phase, the overall deadline of the workflow is distributed across individual tasks. First, the algorithm tries to assign sub-deadlines to all tasks of the (overall) critical path of the workflow such that it can complete before the user's deadline and its execution cost is minimized. Then it finds the *partial critical path* to each assigned task on the critical path and executes the same procedure in a recursive manner. In the latter phase, the planner selects the cheapest service for each task such that the task finishes before its sub-deadline.

The remainder of the paper is organized as follows. Section 2 describes our system model, including the application model, the utility Grid model, and the objective function. The PCP scheduling algorithm is explained in Section 3 followed by an illustrative example. 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 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, respectively.

A utility Grid model consists 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 cost, for our scheduling model. The cost 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 [15], that are beyond our discussion. Besides, there is another source of time and money consumption: transferring data between tasks. We assume $TT(e_{i,j}, r, s)$ and $TC(e_{i,j}, r, s)$ to be the estimated transfer time and transfer cost of sending the required data along $e_{i,j}$ from service $r$ (processing task $t_i$) to service $s$ (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)[16]. In a utility Grid, the 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 advanced 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 element in our model is the performance criterion. 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 criterion is to minimize the execution cost of the workflow while completing the workflow before the user specified deadline.

## 3 The Partial Critical Paths Algorithm

In this section, we first elaborate on the PCP scheduling algorithm, then compute its time complexity, and finally demonstrate its operation through an illustrative example.

## 3.1 Main Idea

The proposed algorithm has two main phases: Deadline Distribution and Planning. In the first phase, the overall deadline of the workflow is distributed over individual tasks, such that if each task finishes before its sub-deadline then the whole workflow finishes before the user defined deadline. In the second phase, the planner selects the cheapest service for each task while meeting its sub-deadline.

Our main contribution in this paper is the deadline distribution algorithm which is based on a Critical Path (CP) heuristic. Critical path heuristics are widely used in workflow scheduling. The *critical path* of a workflow is the longest execution path between the entry and the 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 services that process them earliest, in order to minimize the execution time of the entire workflow. Our deadline distribution algorithm is based on a similar heuristic, but it uses the critical path to distribute the overall deadline of the workflow across the critical nodes. After this distribution, each critical node has a sub-deadline which can be used to compute a sub-deadline for all of its parent nodes, i.e., its (direct) predecessors in the workflow. Then we can carry out the same procedure by considering each critical node in turn as an exit node with its sub-deadline as a deadline, and creating a *partial critical path* that ends in the critical node and that leads back to an already assigned node, i.e., a node that has already been given a sub-deadline. In the *Partial Critical Paths* (PCP) algorithm, this procedure continues recursively until all tasks are successfully assigned a sub-deadline. Finally, the planning algorithm schedules the tasks according to their sub-deadlines. In the following sections, we elaborate on the details of the PCP algorithm.

## 3.2 Basic Definitions

In our PCP scheduling algorithm, we want to find the critical path of the whole workflow, and all 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 planning algorithm.

For each unscheduled task $t_i$ we define its Earliest Start Time, $EST(t_i)$, as the earliest time at which $t_i$ can start its computation, regardless of the actual service that will process the task (which will be determined during planning). Clearly, it is not possible to compute the exact $EST(t_i)$, because a Grid is 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, the minimum execution and data transmission time is selected. The Minimum Execution Time, $MET(t_i)$, and the Minimum Transmission Time, $MTT(e_{i,j})$, are defined as follows:

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

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

Having these definitions, we can compute $EST(t_i)$ as follows:

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

Also, we define for each unscheduled task $t_i$ its Latest Finish Time $LFT(t_i)$ as the latest time at which $t_i$ can finish its computation such that the whole workflow can still finish before the user-defined deadline, $D$. Once again, it is impossible to compute $LFT(t_i)$ exactly and we have to compute it according to the approximate execution and data transmission time as follows:

**Algorithm 1** The PCP Scheduling Algorithm

```
1: procedure ScheduleWorkflow(G(T, E), D)
2:     request available services for each task in G from GMD
3:     add t_entry, t_exit and their corresponding edges to G
4:     compute MET(t_i) for each task according to Eq. 1
5:     compute MTT(e_{i,j}) for each edge according to Eq. 2
6:     compute EST(t_i) for each task in G according to Eq. 3
7:     compute LFT(t_i) for each task in G according to Eq. 4
8:     sub-deadline(t_entry) ← 0, sub-deadline(t_exit) ← D
9:     mark t_entry and t_exit as assigned
10:    call AssignParents(t_exit)
11:    call Planning(G(T, E))
12: end procedure
```

$$\begin{aligned} LFT\left(t_{exit}\right) &= D \\ LFT\left(t_i\right) &= \min_{t_c \in t_i's\ children} LFT\left(t_c\right) - MET\left(t_c\right) - MTT\left(e_{i,c}\right) \end{aligned} \tag{4}$$

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

## 3.3 The Workflow Scheduling Algorithm

Algorithm 1 shows the pseudo-code of the overall PCP algorithm for scheduling a workflow. In line 3, two dummy nodes $t_{entry}$ and $t_{exit}$ are 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 4 - 7, a sub-deadline is assigned to the nodes $t_{entry}$ and $t_{exit}$ (line 8), and they are marked as assigned (line 9). An *assigned* node is defined as a node that has already a sub-deadline, and clearly a node without a sub-deadline is called *unassigned*. As can be seen, the sub-deadline 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 user's deadline. The most important part of the algorithm is the last two lines. In line 10 the procedure *AssignParents* is called for $t_{exit}$. This procedure assigns sub-deadlines to all unassigned parents of its input node. As it has been called for $t_{exit}$, i.e., the last node of the workflow, it will assign sub-deadlines to all workflow tasks. Therefore the *AssignParents* algorithm is responsible for distributing the overall deadline among the workflow tasks. Finally, in line 11 the procedure *Planning* is called to select a service for each task according to its sub-deadline.

## 3.4 The Parents Assigning Algorithm

The pseudo-code for *AssignParents* is shown in Algorithm 2. This algorithm receives an assigned node as input and tries to assign a sub-deadline to all of its parents (the while loop from line 2 to 14 ). First, *AssignParents* tries to find the *Partial Critical Path* of unassigned nodes ending at its input node and starting at one of its predecessors that has no unassigned parent. For this reason, it uses the concept of *Critical Parent*.

**Definition 1** *The Critical Parent of a node $t_i$ is the unassigned parent of $t_i$ that has the latest data arrival time at $t_i$, that is, it is the parent $t_p$ of $t_i$ for which $EST\left(t_p\right) + MET\left(t_p\right) + MTT\left(e_{p,i}\right)$ is maximal.*

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

**Definition 2** *The Partial Critical Path of node $t_i$ is:*

**Algorithm 2** Assigning Deadline to the Parents Algorithm

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

i empty if $t_i$ does not have any unassigned parents.

ii consists of the Critical Parent $t_p$ of $t_i$ and the Partial Critical Path of $t_p$ if has any unassigned parents.

Algorithm 2 begins with the input node and follows the critical parents until it reaches a node that has no unassigned 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 critical path of the complete workflow graph.

Then the algorithm calls procedure *AssignPath* (line 8), which receives a path (an ordered list of nodes) as input, and assigns sub-deadlines to each node on the path before its latest finish time. We elaborate on this procedure in the next sub-section. Note that when a sub-deadline is assigned to a task, the EST of its unassigned successors, and the LFT of its unassigned predecessors may change (according to the Eq. 3 and 4). For this reason, the algorithm updates these values for all tasks of the path in the next loop. Finally, the algorithm starts to assign sub-deadlines to the parents of each node on the partial critical path, from the beginning to the end, by calling *AssignParents* recursively (lines 9-13).

## 3.5 The Path Assigning Algorithm

The *AssignPath* algorithm receives a path as input and assigns sub-deadlines to each of its nodes, for which we propose three policies below. In these policies we try to create an estimated schedule for the path and then use it to determine the sub-deadline of each task on the path. As this is just an estimation and not a real schedule, we do not consider the free time slots of the services in order to speed up our algorithms.

### 3.5.1 Optimized Policy

In this policy, we try to find the cheapest schedule that can execute each task of the path before its latest finish time. Then we use this primary schedule to assign sub-deadlines to the tasks of the path. This policy is shown in Algorithm 3, and it is based on a *Backtracking* strategy. It starts from the first task on the path and moves forward to the last task, at each step selecting the next slower available service for the current task (line 5). Therefore, the services for each task are examined form the fastest to the slowest one. If there is no available untried service for a task left, or assigning the current task $t$, to its next slower service $s$ is not an *admissible* assignment, then the algorithm backtracks to the previous task on the path and selects another service for it (lines 6-8). We call an assignment *admissible* if task $t$ can be finished on service $s$ before the task's latest finish time, i.e., if $EST(t) + ET(t, s) \leq LFT(t)$.

**Algorithm 3** Optimized Path Assigning Algorithm

```
1: procedure AssignPath(path)
2:     best ← null
3:     t ← first task on the path
4:     while (t is not null) do
5:         s ← next slower service ∈ S_t
6:         if (s = ï£¡ or assigning t to s is not admissible) then
7:             t ← previous task on the path and continue while loop
8:         end if
9:         if (t is the last task on the path) then
10:            if (current schedule has a lower cost than best) then
11:                set this schedule as best
12:            end if
13:            t ← previous task on the path
14:        else
15:            t ← next task on the path
16:        end if
17:    end while
18:    if (best is null) then
19:        set sub-deadline(t)=EST(t)+MET(t) for all tasks t on the path
20:    else
21:        set EST and sub-deadline according to best for all tasks ∈ path
22:    end if
23:    mark all tasks of the path as assigned
24: end procedure
```

In the next lines, the algorithm checks if the current task is the last task on the path and has a lower cost than currently best assignment, it is recorded as the current best assignment. At the end of the while loop (line 18), the algorithm checks whether a schedule has been found or not, because there is a chance that some tasks of the path cannot be scheduled before their LFTs. This happens because we have computed the primary ESTs using METs and MTTs, which is an ideal schedule and (almost) does not exist in the real world. So if a task has a very tight LFT (near its current finish time using ideal (minimum) execution times), then we cannot find an estimated schedule for it. In this case, we just use the task's EST+MET as the sub-deadline for that task (line 19). Remember that this is not a real schedule, so we can fix this problem in the planning phase.

At the end, there may be an extra time, i.e., the difference between the LFT of the last task and its sub-deadline, which can be added to the sub-deadlines of the tasks on the path. When this extra time is less than a minimum, we simply add it to the last task's sub-deadline. But if its value is significant, we distribute it over the path's tasks, in proportion to their transfer time plus execution time. Our experiments show that this distribution has a positive effect on the performance of the algorithm. Although we do not explicitly specify this distribution in the path assigning algorithms (Algorithms 3-5), we use it in all three of them.

The main drawback of this policy is its exponential time complexity. Suppose the path length (number of nodes on the path) is $l$, and the maximum number of potential services for a single task is $m$. Then the time complexity of this algorithm is $O(l^m)$. In addition, we transformed the uninformed backtracking search to an informed $A^*$ search [17] (we skip the details for the sake of brevity), which considerably improved the average computation time of the algorithm, although the time complexity in the worst case remains the same.

### 3.5.2 Decrease Cost Policy

This policy is based on a *Greedy* method 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. In this policy, we first assign the fastest service to each task on the path. Obviously this is the most expensive schedule. Then we try 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 reassigned to a cheaper service, we first compute the Cost

**Algorithm 4** Decrease Cost Path Assigning Algorithm

```
1: procedure AssignPath(path)
2:     cur ← assign the fastest service to each task of the path
3:     compute CDR(t_i) for each task of the path according to Eq. 5
4:     repeat
5:         t* ← null
6:         for all (t_i ∈ path) do
7:             if (CDR(t_i) > CDR(t*) and t_i is replaceable) then
8:                 t* ← t_i
9:             end if
10:        end for
11:        if (t* is not null) then
12:            update cur by assigning t* to the next slower service
13:            update CDR(t*)
14:        end if
15:    until (t* is null)
16:    if (there is an inadmissible assignment in cur) then
17:        set sub-dealine(t)=EST(t)+MET(t) for all tasks t on the path
18:    else
19:        set EST and sub-deadline according to cur for all tasks ∈ path
20:    end if
21:    mark all nodes of the path as assigned
22: end procedure
```

Decrease Ratio, CDR, which is defined as follows:

$$CDR(t_i) = \frac{TEC(t_i, cs) - TEC(t_i, ns)}{TET(t_i, ns) - TET(t_i, cs)} \tag{5}$$

where $cs$ is the current service that has been assigned to the task $t_i$ and $ns$ is the next slower service than the current one for $t_i$. The Total Execution Time of the task $t$ on service $s$, $TET(t, s)$, is the sum of the execution time of $t$ on $s$, $ET(t, s)$, plus the total required transfer time between $t$ and its parent and child on the path (except for the first/last task that has no parent/child). The Total Execution Cost of task $t$ on service $s$, $TEC(t, s)$, is defined in a similar manner.

The CDR of a task $t_i$ shows how much it will be execute cheaper in expense of taking one unit of time longer. Then task $t^*$ is selected such that it has the maximum CDR and it is *replaceable*, i.e., assigning it to the next slower service is an admissible assignment. Finally, $t^*$'s current service is changed to the next slower service. Algorithm 4 shows this policy.

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

### 3.5.3 Fair Policy

This policy tries to distribute the path's sub-deadline across the nodes of the path in a fair manner. For this reason, it first schedules the path by assigning each task to its fastest service. Then, starting from the first task towards the last task, it substitutes the assigned service with the next slower service, without exceeding the task's LFT. This procedure continues iteratively until no substitution can be made. The policy is 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 $O(l.m)$.

**Algorithm 5** Fair Path Assigning Algorithm

```
1: procedure AssignPath(path)
2:     cur ← assign the fastest service to each task of the path
3:     repeat
4:         for all (t_i ∈ Path) do
5:             if (assigning t_i to the next service is admissible) then
6:                 update cur by assigning t_i to the next slower service
7:             end if
8:         end for
9:     until (no change is done)
10:    if (there is an inadmissible assignment in cur) then
11:        set sub-dealine(t)=EST(t)+MET(t) for all tasks t on the path
12:    else
13:        set EST and sub-deadline according to cur for all tasks ∈ path
14:    end if
15:    mark all nodes of the path as assigned
16: end procedure
```

## 3.6 The Planning Algorithm

In the planning phase, we try to select the best service for each task of the workflow to create an optimized schedule that ends before the deadline and has the minimum overall cost. In the deadline distribution phase, each task was assigned a sub-deadline. If we schedule each task such that it finishes before its sub-deadline, then the whole workflow will finish before the user's deadline. Our algorithm is based on a *Greedy* strategy that tries to create an optimized global solution by making optimized local decisions. At each stage it selects a ready task, i.e., a task all of whose parents have already been scheduled, and then assigns it to the cheapest service which can execute it before its sub-deadline. So the selected service for a ready task $t_i$, $SS(t_i)$, is the service $s \in S_i$ for which

$$EC(t_i, s) + \sum_{t_p \in t'_i s\, parents} TC(e_{p,i},\, SS(t_p),\, s)$$

is minimized subject to the condition that

$$AST(t_i, s) + ET(t_i, s) \leqslant \text{sub-deadline}\,(t_i) \tag{6}$$

where the Actual Start Time of $t_i$ on $s$, $AST(t_i, s)$, is the maximum between the latest data arrival time of the parents of $t_i$ to the service$s$, and the start time of the suitable free time slot on the service $s$.

It is possible that no service can execute $t_i$ before its sub-deadline, because the sub-deadlines are just estimated schedules and they do not consider the actual free time slots on the services. In that case, we just select the service with the minimum finish time, i.e., $SS(t_i)$ is the service $s \in S_i$ for which

$$AST(t_i, s) + ET(t_i, s) \tag{7}$$

is minimized. Of course, this delay must be compensated in the following selections, as soon as possible. The *Planning* algorithm is shown in Algorithm 6.

## 3.7 Time Complexity

To compute the time complexity of our proposed algorithm, suppose that *ScheduleWorkflow* has received a workflow G(T,E) as input with $n$ tasks and $e$ arcs. Besides, we assume the maximum number of available services for each task is$m$, and $l$ is the length of the longest path between entry and exit tasks. As $G$ is a directed acyclic graph, the maximum number of arcs is $\frac{(n-1)(n-2)}{2}$, so we can assume that $e \simeq O(n^2)$. The

**Algorithm 6** Planning Algorithm

```
1: procedure PLANNING(G(T, E))
2:     Queue ← t_entry
3:     while (Queue is not empty) do
4:         t ← delete first task from Queue
5:         query available time slots for each service from related GSPs
6:         compute SS(t) according to Eq. 6 and 7
7:         AST(t) ← the actual start time of t on SS(t)
8:         make advance reservation of t on SS(t)
9:         for all (t_c ∈ children of t) do
10:            if (all parents of t_c are scheduled) then
11:                add t_c to Queue
12:            end if
13:        end for
14:    end while
15: end procedure
```

most time consuming part of *ScheduleWorkflow* is the deadline distribution phase, i.e., calling *AssignParents*. Nevertheless, we first compute the time complexity of other (main) parts of the algorithm as follow:

- Line 4 (computing METs):$O(n.m)$ =$O(n^3)$
- Line 5 (computing MTTs): $O(e.m^2) = O(n^2.m)$
- Line 6 (computing ESTs):$O(n + e) = O(n^2)$
- Line 7 (computing LFTs): $O(n + e) = O(n^2)$
- Line 11 (*Planning*): $O(n.m.e) = O(n^3.m)$

For the *Planning* algorithm, we should try all services for each task to find the cheapest service that finishes the task before its sub-deadline. In each try, we should compute the actual start time of the task on that service which requires to consider all parent tasks (and their arcs). So the overall time complexity is $O(n.m.e)$.

The *AssignParents* algorithm is a recursive procedure. In the first place, it is called for the exit task and then it calls itself for all of the workflow's tasks. The algorithm has a while loop (lines 2-14) that processes all incoming arcs of each node (task), so it will process all workflow's arcs. Inside the while loop, first it computes partial critical path which its time complexity is $O(l)$. Then it calls *AssignPath* which its time complexity depends on the selected policy. As the time complexity of *AssignPath* depends on $l$ and $m$, let consider it as $g(l, m)$. Therefore, the time complexity of the *AssignParents* is $O(e.l + e.g(l, m))$. Remember that the fastest policy for *AssignPath* was the Fair policy which its time complexity is $O(l.m)$, so we can omit $e.l$ part of the time complexity against the most time consuming part, i.e., $e.g(l, m)$. If we replace $e$, then we have the final time complexity as $O(n^2.g(l, m))$. Note that *AssignParents* also updates the EST of all unassigned successors, and the LFT of all unassigned predecessors of each node after assigning a sub-deadline to it. In the worst case, a node has n-1 unassigned successors and predecessors, so the time complexity of updating ESTs and LFTs for all nodes will be $O(n^2)$ that can be omitted against the bigger part of the time complexity.

Having the time complexity of our three *AssignPath* policies, the time complexity of *AssignParents* will be $O(n^2.l^{\mathrm{m}})$,$O(n^2.l^2.m)$ and $O(n^2.l.m)$, respectively. Now, consider the parameter $l$ and 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 *AssignParents* will be $O(n^{\mathrm{m}})$, $O(n^4.m)$ and $O(n^3.m)$, which is also the time complexity of the whole PCP algorithm.

On the other hand, if we consider real workflows (like realistic workflows we use 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, particularly for the structured workflows. When we consider large workflows,

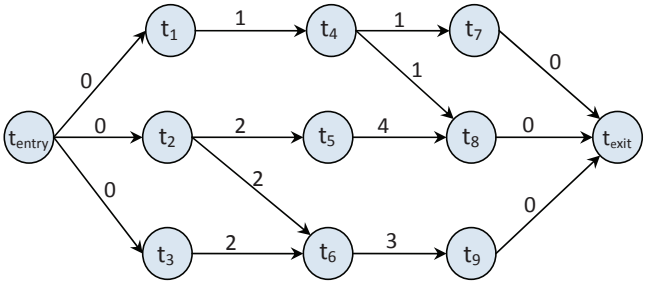



Figure 1: A sample workflow

| Service | Time | Cost | Service | Time | Cost |
|---|---|---|---|---|---|
| $S_{1,1}$ | 6 | 10 | $S_{5,3}$ | 12 | 5 |
| $S_{1,2}$ | 8 | 8 | $S_{6,1}$ | 8 | 12 |
| $S_{1,3}$ | 10 | 5 | $S_{6,2}$ | 12 | 6 |
| $S_{2,1}$ | 5 | 8 | $S_{6,3}$ | 20 | 4 |
| $S_{2,2}$ | 8 | 5 | $S_{7,1}$ | 5 | 8 |
| $S_{2,3}$ | 12 | 3 | $S_{7,2}$ | 9 | 6 |
| $S_{3,1}$ | 4 | 4 | $S_{7,3}$ | 12 | 4 |
| $S_{3,2}$ | 7 | 3 | $S_{8,1}$ | 5 | 10 |
| $S_{3,3}$ | 10 | 1 | $S_{8,2}$ | 8 | 8 |
| $S_{4,1}$ | 8 | 10 | $S_{8,3}$ | 10 | 5 |
| $S_{4,2}$ | 12 | 5 | $S_{9,1}$ | 6 | 8 |
| $S_{4,3}$ | 15 | 4 | $S_{9,2}$ | 12 | 5 |
| $S_{5,1}$ | 6 | 9 | $S_{9,3}$ | 15 | 3 |
| $S_{5,2}$ | 9 | 8 | | | |

Table 1: Available services for the workflow of Figure 1

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, but it is quite reasonable for many realistic workflows. Having this assumption, we can consider $l$ as a constant in the time complexity computation. In this case, the time complexity of *AssignParents* will be $O(n^2.l^{\mathrm{m}})$, $O(n^2.m)$ and $O(n^2.m)$, respectively. Except for the first one, all of these time complexity is dominated by the *Planning* algorithm's time complexity.

### 3.8 An Illustrative Example

In order to show how the algorithm works, we trace its operation on the sample graph shown in Figure 1. The graph consists of nine tasks from $t_1$ to $t_9$, and two dummy tasks, $t_{entry}$ and $t_{exit}$. There are three different services for each task $t_i$, i.e., $S_{i,1}$, $S_{i,2}$ and $S_{i,3}$ which can execute the task with different QoS. Table 1 shows the execution time and the execution cost of each service. It can be seen that for each task, a faster service costs more than a slower one. Furthermore, we suppose that all services are completely available and that they can provide the services at any desired time. To make the example as simple as possible, we suppose that the estimated data transfer time and cost between two adjacent tasks are fixed, and that they are independent of the selected services for the corresponding tasks. In Figure 1, the number above each arc shows both, the estimated

| | Initial | | Step 1 | | | Step 1.1 | | | Step 2 | | | Step 2.1 | | | Step 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tasks | EST | LFT | EST | LFT | DL | EST | LFT | DL | EST | LFT | DL | EST | LFT | DL | EST | LFT | D |
| $t_1$ | 0 | 20 | 0 | 20 | - | 0 | 20 | - | 0 | 15* | - | 0 | 11* | 10* | 0 | 11 | 1 |
| $t_2$ | 0 | 16 | 0 | 12* | 12* | 0 | 12 | 12 | 0 | 12 | 12 | 0 | 12 | 12 | 0 | 12 | 1 |
| $t_3$ | 0 | 16 | 0 | 12* | - | 0 | 12 | 10* | 0 | 12 | 10 | 0 | 12 | 10 | 0 | 12 | 1 |
| $t_4$ | 7 | 29 | 7 | 29 | - | 7 | 29 | - | 7 | 24* | - | 11* | 24 | 23* | 11 | 24 | 2 |
| $t_5$ | 7 | 26 | 14* | 26 | - | 14 | 26 | - | 14 | 21* | 20* | 14 | 21 | 20 | 14 | 21 | 2 |
| $t_6$ | 7 | 26 | 14* | 26* | 26* | 14 | 26 | 26 | 14 | 26 | 26 | 14 | 26 | 26 | 14 | 26 | 2 |
| $t_7$ | 16 | 35 | 16 | 35 | - | 16 | 35 | - | 16 | 35 | - | 24* | 35 | - | 24 | 35 | 3 |
| $t_8$ | 17 | 35 | 24* | 35 | - | 24 | 35 | - | 24* | 35 | 34* | 24 | 35 | 34 | 24 | 35 | 3 |
| $t_9$ | 18 | 35 | 29* | 35* | 35* | 29 | 35 | 35 | 29 | 35 | 35 | 29 | 35 | 35 | 29 | 35 | 3 |

Table 2: The values of EST, LFT and sub-deadline (DL) for each Step of running the PCP algorithm on the sample workflow of Figure 1

data transfer time and cost between the corresponding tasks. For example, the estimated data transfer time between $t_2$ and $t_5$ is 2 (time units) and it also costs 2 (monetary units) for the user, independent of the selected services for these two tasks. Finally, let the overall deadline of the workflow be 35.

When we call the PCP scheduling algorithm, i.e., Algorithm 1, for the sample workflow of Figure 1, it first computes the Earliest Start Time (EST) and the Latest Finish Time (LFT) for each task of the workflow by assigning them to their fastest service. The initial value of these parameters are shown in Table 2 under the *Initial* column. Then the algorithm sets the sub-deadlines of $t_{entry}$ and $t_{exit}$ to zero and 35, respectively, and marks them as assigned. The next steps are to call the main procedures of the algorithm, *AssignParents* and *Planning*, which will be discussed now.

#### 3.8.1 Calling *AssignParents*

First, the procedure *AssignParents* (Algorithm 2) is called for task $t_{exit}$. As this task has three parents, the while loop in line 2 will be executed three times, which we call Step 1 to Step 3, and each one will be discussed separately. Furthermore, we should choose one of the path assigning policies for this example, which will be the Optimized policy. The new values of EST, LFT and sub-deadline (DL column) of the workflow tasks for each step are given in the related column in Table 2. Note that the appearance of a star mark (*) in front of a cell shows that the value of that cell has been changed over the previous step. Also, the column DL stands for the sub-deadline of each task.

*Step 1:* At first, the *AssignParents* algorithm follows the partial critical parent of $t_{exit}$ to find its partial critical path, which is $t_2$-$t_6$-$t_9$. Then it calls *AssignPath* with the Optimized policy (Algorithm 3) to assign sub-deadlines to these tasks. There are 27 possible service assignments for these three tasks, and among them the assignment of $S_{2,3}$ to $t_2$, $S_{6,2}$ to $t_6$ and $S_{9,1}$ to $t_9$ is the best admissible assignment with minimum cost. This assignment is used to determine the sub-deadline of each task. The next step is to update the EST of all unassigned successors of these three tasks, i.e., $t_5$ and $t_8$, and also the LFT of their unassigned predecessors, i.e., $t_3$. These changes are shown in Table 2 under the Step 1 column. The final step is to call *AssignParents* recursively for all tasks on the path. Since tasks $t_2$ and $t_9$ have no unassigned parents, we just consider calling of *AssignParents* for task $t_6$ in Step 1.1.

*Step 1.1:* When *AssignParents* is called for task $t_6$, it first finds the partial critical path of this task, which is $t_3$. Then it calls *AssignPath* to find the best admissible assignment for $t_3$, which is $S_{3,3}$. Since $t_3$ has no unassigned child or parent, Step 1 finishes here.

*Step 2:* Now, back to task $t_{exit}$, the *AssignParents* tries to find the next partial critical path of this task, which is $t_5$-$t_8$. Then it calls *AssignPath*, which considers all 9 possible assignments for these two tasks and

selects the best admissible assignment as assigning $S_{5,1}$ to $t_5$ and $S_{8,3}$ to $t_8$. The tasks have no unassigned successor, but the algorithm updates the LFT of their unassigned predecessors, which are $t_1$ and $t_4$. Finally, the algorithm calls *AssignParents* for all tasks on the path, $t_5$ has no unassigned parent, so we just consider $t_8$ in Step 2.1.

*Step 2.1:* When *AssignParents* is called for task $t_8$, it finds the partial critical path for it, which is $t_1$-$t_4$, and then calls *AssignPath* which computes the best admissible assignment of this path as assigning $S_{1,3}$ to $t_1$ and $S_{4,2}$ to $t_4$. The tasks have no unassigned predecessors, but the algorithm updates the EST of $t_4$'s child, which is $t_7$. As $t_1$ and $t_4$ have no unassigned parents, Step 2 stops.

*Step 3:* In the final step, *AssignParents* finds the last partial critical path of $t_{exit}$, which is $t_7$. *AssignPath* finds the best admissible assignment as assigning $S_{7,2}$ to $t_7$, and since there is no unassigned parent or child, the algorithm stops.

Note that the value of the DL column is computed according to the finish time of the tasks in the best admissible assignment. By the way, as we discussed in Section 3.5.1, the sub-deadline of the last task on the path can be set to its latest finish time (when the extra time is small). For example, the value of DL for task $t_7$ under the Step 3 column should be set to 35 instead of 33. Nevertheless, we do not change the DLs to make the calculations more clear.

### 3.8.2 Calling *Planning*

In this simple example, the *Planning* algorithm just schedules each task on the same service that was assigned to that task by the *AssignParents* algorithm. There are two reasons for this issue: fixed data transfer times and fully available services. These two assumptions make the estimated schedules of the *AssignPath* (and consequently the *AssignParents*) as real schedules and since they are optimized schedules, the *Planning* algorithm selects the same services. The selected services are shown in the last column of Table 2. The total time is 35 and the total cost is 64, including execution cost (48) and data transfer cost (16).

# 4 Performance Evaluation

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

## 4.1 Experimental Workflows

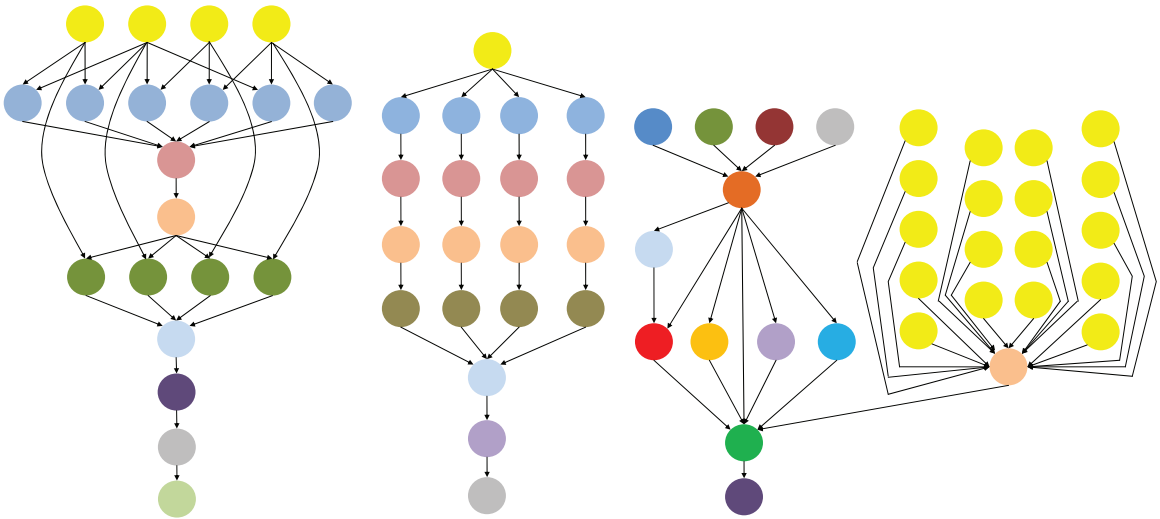
To evaluate a workflow scheduling algorithm, we should measure its performance on some sample workflows. There are two ways to choose these sample workflows:

i Using a random DAG generator to create a variety of workflows with different characteristics.

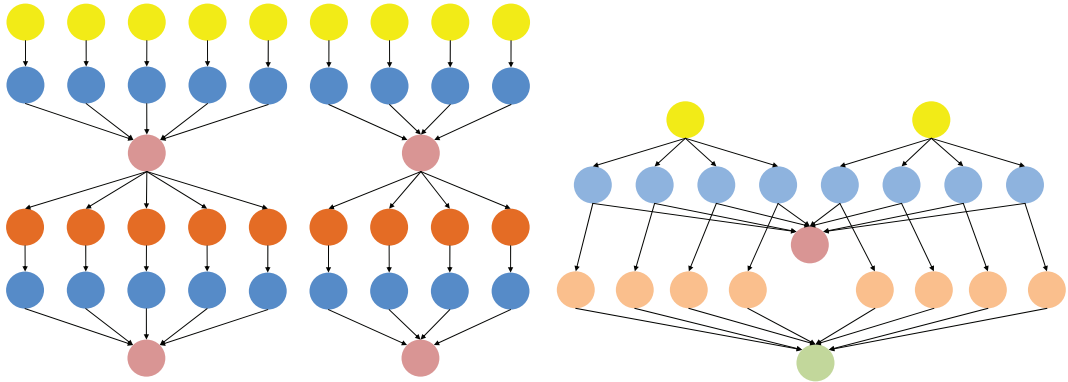
ii Using a library of realistic workflows which are used in the scientific or business community.

Although the latter seems to be a better choice, unfortunately there is no such a comprehensive library available to researchers. One of the preliminary works in this area is done by Bharathi et al. [1]. They study the structure of five realistic workflows from diverse scientific applications, which are:

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



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



(d) LIGO (e) CyberShake

Figure 2: The structure of five realistic scientific workflows [1]

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

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

## 4.2 Experimental Setup

We use GridSim [18] for simulating the utility Grid environment for our experiments. We simulate a multicluster Grid environment, consists of 10 heterogeneous clusters. Each cluster has a random number of nodes between 20 to 100. All nodes of a cluster has the same processor speed, which is determined randomly such that the fastest cluster is 10 times faster than the slowest one. We assume all required services are installed on every cluster, such that all workflow tasks can be executed on each arbitrary cluster. Furthermore, a random cost per second is assigned to each cluster, following that a faster cluster costs more than a slower one. The average inter-cluster bandwidth is a random number between 100 to 512 Mbps, and the data transfer costs are assigned proportional to the bandwidths, i.e., a higher bandwidth costs more than a lower one. The intra-cluster bandwidth is assumed to be 1 Gbps for each cluster and is free. In addition, we assume that all clusters are empty in the beginning.

Finally, we compare the PCP algorithm with the Deadline-MDP, one of the most cited algorithms in this area that has been proposed by Yu et al. [19]. They divide 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 the cost of each partition execution under its sub-deadline constraint.

## 4.3 Experimental Results

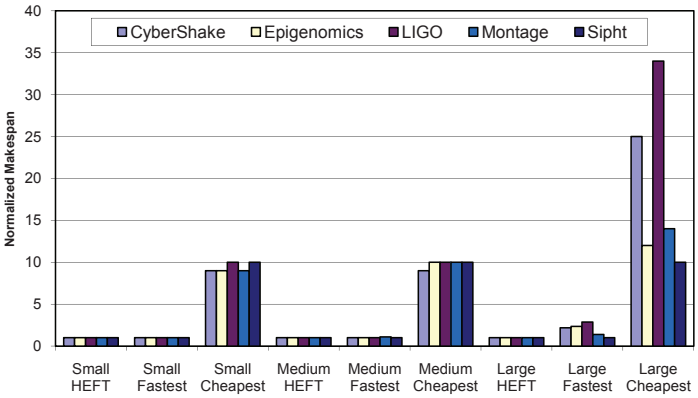
First, to get a better idea of the required time and cost for each workflow application, we simulate their execution using three scheduling algorithms: *HEFT* [11], a well-known makespan minimization algorithm, *Fastest,* which submits all tasks to the fastest cluster, and *Cheapest,* which submits all tasks to the cheapest (and slowest) cluster. Note that the last two algorithms submit all tasks to one cluster (fastest or cheapest), and therefore some tasks may 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 Makespan (NM) and the Normalized Cost (NC) of a workflow execution as follows:

$$NM = \frac{schedule\,makespan}{M_H} \tag{8}$$

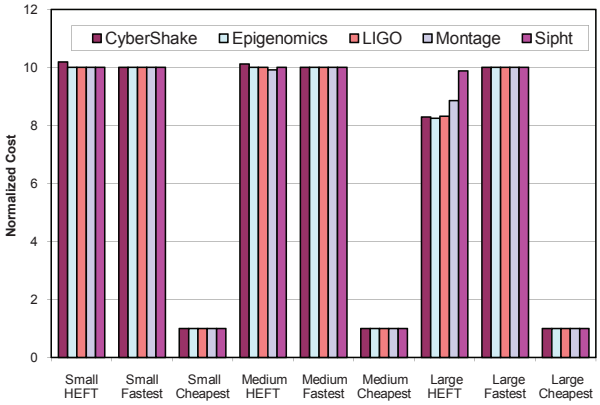
$$NC = \frac{total\ schedule\,cost}{C_C} \tag{9}$$

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 the workflow applications using these three scheduling policies are shown in Figure 3. Obviously, the normalized makespan and the normalized cost are the same for the HEFT and Fastest strategy for small workflows, because the number of tasks is less

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




(a) Normalized Makespan




(b) Normalized Cost

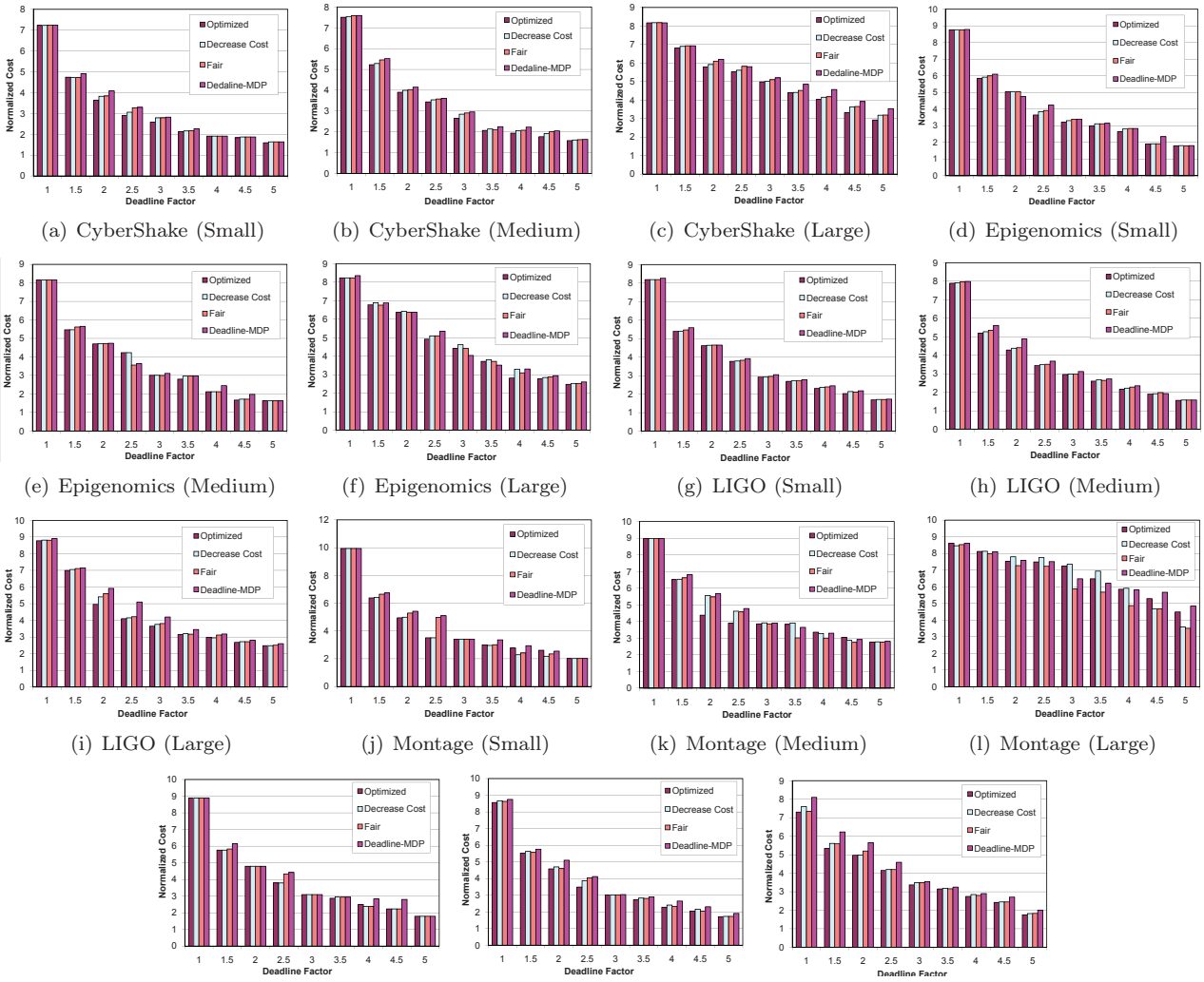
Figure 3: Normalized Makespan and Normalized Cost of scheduling workflows with three scheduling policies: HEFT, Fastest and Cheapest

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.

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* $\alpha$, and we set the deadline of a workflow to the time of its arrival plus $\alpha \cdot M_H$. In our experiments, we let $\alpha$ range from 1 to 5.

Both algorithms successfully scheduled all workflows before their deadlines, even in the case of tight deadlines (small deadline factor). Table 3 shows the average percentage by which the normalized makespan is smaller than the deadline factor for all workflows. It can be seen that both algorithms almost use all available deadline to minimize the execution cost for LIGO and CyberShake workflows, i.e., their average difference percentages are less than 1%. This is almost the case for Montage, but for Epigenomics and SIPHT, the Deadline-MDP algorithm has high average difference percentages, e.g., 3.07% for medium Epigenomics and 5.99% for small SIPHT. Although, all three policies of the PCP algorithm also have rather high difference percentages for the small SIPHT.

Figures 4 shows the cost of scheduling all workflows with the PCP (including three path assigning policies)




(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 4: Normalized Cost of scheduling workflows with the PCP and Deadline-MDP algorithms

| | | PCP | | | Deadline |
|---|---|---|---|---|---|
| | | Optimized | DC | Fair | -MDP |
| CyberShake | small | 0.51 | 0.36 | 0.67 | 0.93 |
| | medium | 0.17 | 0.01 | 0.18 | 0.06 |
| | large | 0.34 | 0.39 | 0.21 | 0.32 |
| Epigenomics | small | 0.19 | 0.19 | 0.40 | 2.94 |
| | medium | 1.17 | 1.36 | 1.72 | 3.07 |
| | large | 0.52 | 1.11 | 1.06 | 2.09 |
| LIGO | small | 0.20 | 0.39 | 0.21 | 0.47 |
| | medium | 0.08 | 0.09 | 0.18 | 0.15 |
| | large | 0.05 | 0.30 | 0.25 | 0.49 |
| Montage | small | 0.49 | 0.41 | 0.61 | 0.46 |
| | medium | 0.03 | 0.74 | 0.39 | 0.88 |
| | large | 0.21 | 0.87 | 0.35 | 1.17 |
| SIPHT | small | 2.24 | 3.53 | 3.10 | 5.99 |
| | medium | 1.19 | 0.89 | 1.14 | 2.44 |
| | large | 0.04 | 0.16 | 0.05 | 0.40 |

Table 3: The average percentage by which the Normalized Makespan is smaller than the deadline factor ($\alpha$)

| | Small | Medium | Large |
|---|---|---|---|
| CyberShake | 5.56 | 8.13 | 9.04 |
| Epigenomics | 6.46 | 3.75 | 2.92 |
| LIGO | 3.65 | 6.78 | 10.83 |
| Montage | 8.48 | 5.44 | 0.04 |
| SIPHT | 7.23 | 9.32 | 12.26 |

Table 4: Average cost decrease in percent of the PCP (Optimized policy) over the Deadline-MDP

and the Deadline-MDP algorithms. A quick look at Figure 4 shows that the results for small and medium size workflows are almost similar. In all of them, both algorithms have (almost) the same normalized cost (about 2) for a relaxed deadline, i.e., deadline factor equal to 5. 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 $C_C$ for all small and medium size workflows. The only exception is the medium Montage workflow (Figure 4(k)). But for the large workflows things are completely different for some workflows. The only large workflow that maintains the same results as the smaller ones is the SIPHT workflow, while the Montage has the worst performance. 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. Figure 4 also shows that the optimized policy has the best performance (lowest cost) among the three policies for the PCP algorithm in most cases. It also outperforms the Deadline-MDP in many cases. Table 4 shows the average cost decrease of using the PCP algorithm with the Optimized policy over the Deadline-MDP algorithm for each workflow.

For CyberShake, LIGO and SIPHT workflows, the PCP scheduling algorithm with the Optimized policy has the best performance, while the Decrease Cost policy has a very close performance. The Fair policy has a lower performance, still it performs better than the Deadline-MDP. Table 4 shows that the PCP algorithm has a very promising result over the Deadline-MDP for these workflows.

For the Epigenomics workflow, Deadline-MDP has a better performance than PCP in some cases, which results in a small average cost decrease for the medium and large sizes (Table 4). The problem of the PCP algorithm with this workflow is about its structure. Considering the Epigenomics structure in Figure 2(b) shows

| | Optimized | Decrease Cost | Fair |
|---|---|---|---|
| CyberShake | 156 | 141 | 140 |
| Epigenomics | 28371 | 31 | 31 |
| LIGO | 6552 | 47 | 31 |
| Montage | 48215 | 141 | 141 |
| SIPHT | 1388 | 140 | 125 |

Table 5: Maximum computation time of different path assigning policies for the large size workflows (ms)

that it consists of multiple parallel pipelines operating on distinct chunks of data. At the beginning, when the PCP finds the critical path of the whole workflow, it obviously consists of the entry task, one of the parallel pipes (with four tasks), plus the final three tasks of the workflow. Then PCP tries to find the best schedule for this critical path, without considering the other parallel pipes between the first and the sixth tasks. But if we consider the other parallel pipes, likely it is better to assign longer sub-deadlines to these four tasks, because the other parallel pipelines also benefit from this extra time and the overall cost is reduced. We are working on a modification of the PCP algorithm to solve this problem for highly parallel workflows.

The large size Montage has the worst performance for all algorithms, i.e., the cost hardly decreases with the deadline increase. The results show that when we increase the deadline about 5 times, the cost decreases about half of the initial value for the large size Montage. Furthermore, the performance of the PCP algorithm with the Optimized policy decreases from the small size instance to the large size, such that in the large size instance, it has worse performance than the Deadline-MDP in some cases and the overall performance is almost the same (see Table 4). To find the reason, consider the Montage structure in Figure 2(a). The overall critical path of this workflow consists of 9 tasks, which are assigned sub-deadlines first by the Otimized policy according to the best possible schedule for this path. For the small size Montage, these sub-deadlines are respected in the Planning phase. However, for the large Montage, there are many tasks before the third task on the overall critical path which overflow the faster clusters and force the Planning algorithm to schedule them on slower clusters. As a result, the third task on the critical path cannot finish on time, and this delay is propagated to its successors, which decreases the final schedule performnce. Fortunately, Fair policy has a very promising performance for large Montage, such that its average cost decrease percentage over Deadline-MDP is 12.07. This policy also has a good performance for the medium size Montage.

## 4.4 Computation Time

In this section we try to answer some questions like what is the actual computation time of the PCP algorithm?, and what is the impact of different path assigning policies, different workflow types and different deadlines on the computation time of this algorithm? Table 5 shows the maximum computation time (in milliseconds) of the algorithm for the large size workflows, using three different policies. The computation time of the small and medium size workflows have a similar pattern with smaller values (except for the Montage).

For the CyberShake workflow, there is no significant difference among the three policies. This is not surprising, because this workflow has a small length $l$ of the longest path between the entry and exit tasks, equal to 4, which is the most important factor in the time complexity of the Optimized policy. Furthermore, the overall deadline of the workflow has no impact on the computation time, i.e., the computation time is almost the same for $\alpha = 1$ to $\alpha = 5$. This is the only workflow with such properties.

But for Epigenomics, there is a huge difference between the Optimized policy and the other two policies, because it has a relatively long $l$ equal to 8. The overall deadline influences the computation time of the Optimized policy, i.e., it is much lower for a tight deadline than a loose deadline. The reason is that for a tight deadline only the fast services can be used, and most of the slower services are not considered. For example, the computation time for the large Epigenomics is about 35 ms for $\alpha = 1$, 1,459 ms for $\alpha = 1.5$ and 28,731

for $\alpha = 5$. LIGO and SIPHT workflows have the same properties as Epigenomics, except that the difference between the Optimized and the other two policies is much lower, because of their smaller $l$, which is equal to 6.

The Montage workflow has the longest computation time for the Optimized policy, with its $l$ equal to 9. Furthermore, it is the only workflow whose computation time for the small and medium size instances are almost equal to that of the large one, i.e., its maximum computation time for the small instance is 44,123 ms and for the medium instance is 45,056 ms. To find the reason, we should look at the Montage structure in Figure 2(a). The most time consuming part is to find the overall critical path of the workflow (with 9 tasks) and assigning sub-deadline to its tasks. This phase is common between all three sizes. The next phase is to find the partial critical paths and assigning sub-deadline to them. Figure 2(a) shows that the next partial critical paths consist of one or two tasks, so assigning sub-deadlines to them is easy and fast. Therefore, althought the large size workflow has more tasks than the small and medium size instances, assigning sub-deadlines to these extra tasks is not time consuming and the overall computation times are almost the same. Finally, the computation time is highly dependent on the overall deadline like Epigenomics.

Table 5 shows that the algorithm has an acceptable computation time, even for the Optimized policy with its exponential time complexity. Obviously, this is the case for our sample realistic workflows for which the maximum $l$ is 9. For a workflow with a large $l$, or in a Grid environment with a large number of resources (large $m$), the Optimized policy cannot be used. Fortunately, Figure 4 shows that the other two policies have a close (and sometimes better) performance to the Optimized policy and can be used in such cases.

# 5 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 mentioned the Deadline-MDP proposed by Yu et al. [19]. Yuan et al. [20] proposed the DET (Deadline Early Tree) algorithm which is based on a similar method. First, they create a primary schedule as a spanning tree which is called *Early Tree*. Then, they try to distribute the whole deadline among workflow tasks by assigning a time window to each task. This is achieved in two steps: first a dynamic programming approach assigns time windows to critical tasks, then an iterative procedure finds a time window for each non-critical task. Finally, a local optimization method tries to minimize the execution costs according to the dedicated time windows.

Sakellariou et al. [21] proposed two scheduling algorithms for a different performance criterion: minimizing the execution time under budget constraint. 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 its cheapest resource, and then try to refine the schedule to shorten the execution time under budget constraint.

Prodan et al. [22] proposed a bi-criteria scheduling algorithm that follows a different approach to the optimization problem of two arbitrary independent criteria, e.g., execution time and cost. In the Dynamic Constraint Algorithm (DCA), the end user defines three important factors: the *primary* criteria, the *secondary* criteria and the *sliding constraint* which determines how much the final solution can vary from the best solution for the primary criterion. The DCA has two phases: 1) primary scheduling finds the preliminary solution which is the best solution according to the primary criterion, 2) secondary scheduling optimizes the preliminary solution for the secondary criterion while keeping the primary criterion within the predefined sliding constraint. The primary scheduling algorithm depends on the type of primary criteria, e.g., HEFT is used for the execution time. For the secondary scheduling, the authors model the problem as a multiple choice knapsack problem and solve it using a heuristic based on a dynamic programming method.

Duan et al. [23] proposed two algorithms based on Game Theory for the scheduling of $n$ independent workflows. The first one called Game-quick, tries to minimize the overall makespan of all workflows. The second algorithm called Game-cost, tries to minimize the overall cost of all workflows, while meeting each workflow's deadline. Brandic et al. [24] solved the problem of multi-objective workflow scheduling using Integer

Programming. To transform the multi-objective problem to a single-objective one, the user should assign a weight to each QoS parameter and the algorithm tries to optimize the weighted sum of the QoS parameters. In Afzal et al. [25] the problem of cost minimization under deadline constraint is solved using Mixed-Integer Non-Linear Programming (MINLP). They also used queueing theory to take the inherent unpredictability of the Grid into account.

Meta-heuristic methods are also used to tackle this problem. Although these methods have a good performance, but usually are more time consuming than the heuristic ones. Yu et al. [3] used Genetic Algorithm to solve both problems: cost optimization under deadline constraint, and execution time optimization under budget constraint. In another work, Yu et al. [26] used three multi-objective evolutionary algorithms, i.e., NSGAII, SPEA2 and PAES for this problem. In this work, the user can specify his desired deadline and maximum budget and the algorithm proposes a set of alternative trade-off solutions meeting the user's constraints from which he can select the best one according to his needs. In a similar research, Talukder et al. [27] proposed a Multiobjective Differential Evolutionary algorithm which generates a set of alternative solutions for the user to selecet. Chen and Zhang [28] proposed an Ant Colony Optimization (ACO) algorithm, which considers three QoS parameters, i.e., time, cost and reliability. They enable the users to determine constraints for two of these parameters, and the algorithm finds the optimized solution for the third parameter while meeting those constraints. In Quan et al. [29], the performance of six different algorithms, i.e., Tabu Search, Simulated Annealing, Iterated Local Search, Guided Local Search, Genetic Algorithm and Estimation of Distribution Algorithm are compared together in solving the problem of cost optimization under deadline constraint. Finally, Tao et al. [30] considered a multi dimensional QoS parameters like time, cost, reliability, availability, reputation and security, and tried to optimize the weighted sum of these parameters using Particle Swarm Optimization (PSO).

Among the current workflow management systems, Amadeus [31] supports QoS-aware workflow definition, scheduling and execution. In the definition phase, it provides a graphical user interface that enables user not only to define his workflow, but also to specify local (for one task) and/or global (for the whole workflow) QoS constraints. For the scheduling phase, it has two different strategies: static scheduling uses Integer Programming (same as [24]) and dynamic scheduling that selects services during the execution when a task is ready to start. Another example is ICENI [32] which uses several algorithms like Game Theory and Simulated Annealing to optimize a single *benefit function,* e.g., time or cost. Furthermore, a number of algorithms are implemented and tested on existing Grid environment, e.g., the previously mentioned algorithm in [22] is implemented as part of the ASKALON.

# 6 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 named Partial Critical Paths (PCP) for workflow scheduling in utility Grids that minimizes the total execution cost while meeting a user-defined deadline. The PCP algorithm has two phases: deadline distribution and planning. In the deadline distribution phase, the overall deadline of the workflow is divided over the workflow's tasks, for which we proposed three different policies, i.e., Optimized, Decrease Cost and Fair. In the planning phase, the best service is selected for each task according to its sub-deadline. We evaluate our algorithm by simulating it with synthetic workflows that are based on real scientific workflows with different structures and different sizes. The results show that 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 modify our algorithm to improve its performance on parallel pipelines. Furthermore, we will extend our algorithm to support Cloud computing models.

## References


[1] 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. 3, 15, 16

[2] 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. 4

[3] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program.*, vol. 14, no. 3,4, pp. 217–230, 2006. 4, 23

[4] 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, no. 3, pp. 255–276, 2008. 4

[5] E. Deelman *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, pp. 219–237, 2005. 4

[6] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the askalon grid environment," *SIGMOD Rec.*, vol. 34, pp. 56–62, 2005. 4

[7] F. Berman *et al.*, "New grid scheduling and rescheduling methods in the grads project," *Int'l J. Parallel Program.*, vol. 33, pp. 209–229, 2005. 4

[8] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *J. of Grid Comput.*, vol. 3, pp. 171–200, 2005. 4

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

[10] Y. K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, 1999. 4

[11] 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, no. 3, pp. 260–274, 2002. 4, 17

[12] R. Bajaj and D. P. Agrawal, "Improving scheduling of tasks in a heterogeneous environment," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 107–118, 2004. 4

[13] 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, no. 4, pp. 399–409, 2008. 4

[14] 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–. 4

[15] 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. 5

[16] 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 J. of Supercomputing*, vol. 36, no. 1, pp. 17–31, 2006. 5

[17] S. Russell and P. Norvig, *Artificial Intelligence, A Modern Approach*, 3rd ed. Prentice Hall, January 2009. 9

[18] R. Buyya and M. Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurr. Comput. : Pract. Exper.*, vol. 14, no. 13, pp. 1175–1220, 2002. 17

[19] 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. 17, 22

[20] Y. Yuan, X. Li, Q. Wang, and X. Zhu, "Deadline division-based heuristic for cost optimization in workflow scheduling," *Information Sciences*, vol. 179, no. 15, pp. 2562 – 2575, 2009. 22

[21] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling workflows with budget constraints," in *Integrated Research in GRID Computing*, ser. CoreGRID Series, S. Gorlatch and M. Danelutto, Eds., 2007, pp. 189–202. 22

[22] R. Prodan and M. Wieczorek, "Bi-criteria scheduling of scientific grid workflows," *IEEE Trans. on Automation Sci. and Eng.*, vol. 7, no. 2, pp. 364 –376, 2010. 22, 23

[23] 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

[24] 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. 22, 23

[25] 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. 23

[26] J. Yu, M. Kirley, and R. Buyya, "Multi-objective planning for workflow execution on grids," in *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, 2007, pp. 10–17. 23

[27] A. K. M. K. A. Talukder, M. Kirley, and R. Buyya, "Multiobjective differential evolution for scheduling workflow applications on global grids," *Concurr. Comput. : Pract. Exper.*, vol. 21, pp. 1742–1756, 2009. 23

[28] 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, no. 1, pp. 29–43, 2009. 23

[29] 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, no. 3, pp. 330–346, 2008. 23

[30] Q. Tao, H. Chang, Y. Yi, C. Gu, and Y. Yu, "Qos constrained grid workflow scheduling optimization based on a novel pso algorithm," in *Eighth International Conference on Grid and Cooperative Computing*, 2009, pp. 153 –159. 23

[31] I. Brandic, S. Pllana, and S. Benkner, "Specification, planning, and execution of qos-aware grid workflows within the amadeus environment," *Concurr. Comput. : Pract. Exper.*, vol. 20, pp. 331–345, 2008. 23

[32] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington, "Workflow Enactment in ICENI," in *UK e-science all-hands meeting, AHM 2004*, 2004, pp. 894–900. 23