QoS-based Scheduling of Workflow Applications on Service Grids

Jia Yu[†], Rajkumar Buyya[†] and Chen Khong Tham[‡]

[†] Grid Computing and Distributed System Laboratory
Dept. of Computer Science and Software Engineering
The University of Melbourne, VIC 3010 Australia
{jiayu, raj}@csse.unimelb.edu.au

[‡] Dept. of Electrical and Computer Engineering
National University of Singapore
10 Kent Ridge Crescent, Singapore 119260
eletck@nus.edu.sg

***Abstract*—Over the last few years, Grid technologies have been enhanced towards a service-oriented paradigm that enables a new way of service provision based on utility computing models, which users consume based on their QoS (Quality of Service) requirements. In such "pay-per-use" service Grids, issues such as resource management and scheduling based on users' QoS constraints are yet to be addressed especially in the context of workflow management systems. In this paper, we propose a QoS-based workflow management system and scheduling algorithm that minimizes execution cost workflow application while meeting timeframe for delivering results. We also attempt to optimally solve the task scheduling problem in branches with several sequential tasks by modeling the branch as a Markov Decision Process and using the value iteration method.**

I. Introduction

Utility computing [13] has emerged as a new service provision model and its services [7] are capable of supporting diverse applications including e-Business and e-Science over a global network. The users utilize the services when they need to, and pay only for what they use. In the recent past, providing utility computing services has been reinforced by service-oriented Grid computing [9] by providing an infrastructure that enables users to consume utility services transparently over a secure, shared, scalable and standard world-wide network environment.

Many Grid applications such as bioinformatics and astronomy require workflow processing in which tasks are executed based on their control or data dependencies. As a result, a number of Grid workflow management systems with scheduling algorithms have been developed by several projects (e.g. Condor DAGMan [16], Askalon [8], GrADS [5], ICENI [10], APST [18], and Pegasus [6][17]). They facilitate workflow application execution on Grids and minimize execution time. However, scheduling workflows based on users' QoS (Quality of Service) requirements (e.g. deadline and budget) has not been addressed in these existing Grid workflow management systems. For a utility service, pricing is dependent on the level of QoS offered. Typically service providers charge higher prices for higher QoS. Therefore, users may not always need to complete workflows earlier than they require. Instead, they prefer to use cheaper services with lower QoS that are sufficient to meet their requirements. Given this motivation, we focus on QoS-based workflow management which attempts to minimize execution cost while satisfying users' QoS requirements.

In this paper, we discuss basic QoS-based workflow management requirements for service Grids and present a novel workflow scheduling method. The objective function of the proposed scheduling algorithm is to develop workflow schedule such that it minimizes the execution cost and yet meet the time constraints imposed by the user. In order to solve scheduling problems efficiently for large-scale workflows, we partition workflow tasks and generate the workflow execution schedule based on the optimal schedules of task partitions. A deadline assignment strategy is also developed to distribute the overall deadline over each partition. We also attempt to solve optimally the scheduling problem for sequential tasks by modeling the branch partition as a Markov Decision Process (MDP) [12], which has proven to be effective for modeling decision problems.

Proposed workflow scheduling approach can be used by both end-users and utility providers. End users can use the approach to orchestrate Grid services, while utility providers can outsource computing resources to meet customers' service-level requirements.

The remainder of the paper is organized as follows. Section II provides an overview of QoS-based workflow management on service grids. We describe our novel workflow scheduling approach in Section III. Experimental details and simulation results are presented in Section IV. Finally, we conclude the paper with directions for further work in Section V.

II. QoS-based Workflow Management system

QoS-based workflow management on service Grids impacts all levels including workflow specification, service discovery and workflow scheduling. In this paper we use the term *service* to mean utility computing service as described before. The architecture of a typical QoS-based workflow management system is shown in Figure 1. The components of the workflow management system are discussed below.

A. *Workflow Specification*

The QoS-based workflow management system allows the user to specify their requirements along with the descriptions of tasks and their dependencies using the workflow specification. In general, QoS constraints express the preferences of users and are essential for efficient resource

allocation. We categorize workflow QoS constraints into task-level and workflow-level constraints. At the task level, as illustrated in Figure 2, QoS constraints are specified with their corresponding tasks. In this scenario, the two QoS constraints, namely time and cost are specified with task A. In contrast, QoS constraints at the workflow level are given for entire workflow execution. In the example shown in Figure 3, the workflow execution is required to be completed before *2005-12-10T04:20:00.000+10:00* at the minimum cost.

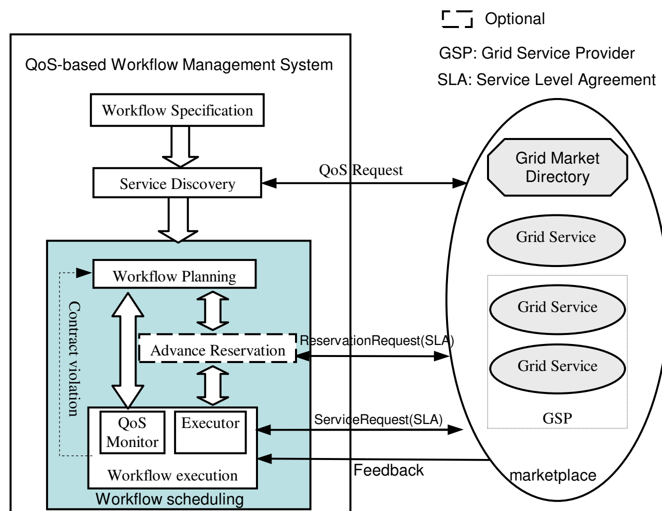


Fig. 1. QoS-based workflow management system architecture.

```
<Workflow>
   <tasks>
      <task name="A">
         <qos-constraints>
            <qos-constraint name="time" value=" "/>
            <qos-constraint name="cost" value=" "/>
         </qos-constraints>
         …..
      </task>
      ……
   </tasks>
   ……
</Workflow>
```

Fig. 2. Task-level QoS specification.

```
<Workflow>
   <qos-constraints>
      <qos-constraint name="time" value="2005-12-10
   T04:20:00.000+10:00"/>
      <qos-constraint name="cost" optimal="on"/>
   </qos-constraints>
   <tasks>
      ……
   </tasks>
   ……
</Workflow>
```

Fig 3. Workflow-level QoS specification.

Users may want to specify QoS constraints, such as deadline and budget, for the overall workflow processing rather than for each task. For instance, users may want the entire workflow execution finished in 2 hours instead of specifying an execution time of 30 minutes for each task. In this paper, we focus on the overall time constraint, i.e. deadline.

B. *Service Discovery and QoS Request*

After submission of the workflow specification, the workflow system needs to discover the appropriate services for processing the tasks. In a complex workflow, different tasks require different types of services. For example, for a biological imaging process, some tasks need to access a genome search service and some other tasks need to access a protein folding service. However, in a service Grid, even for the same type of services, they are deployed by different service providers and are distributed across multiple administrative domains. In addition, every service has its own local policy for different users, such as authorization and pricing. The workflow system should be able to query a Grid information service such as a grid market directory and generate a list of available services for every task for the user of the workflow.

In a service Grid, the QoS attributes of services for processing the same task is diverse. Different service providers can offer different QoS. One service provider also can offer various QoS levels for satisfying different users' requirements. The pricing for the services is usually closely related to the QoS provided. However, some users may have priority in terms of service order, execution time and price from certain service providers. In addition, service providers may adjust the service price based on peak and off-peak periods in order to enhance the utilization of their resources.

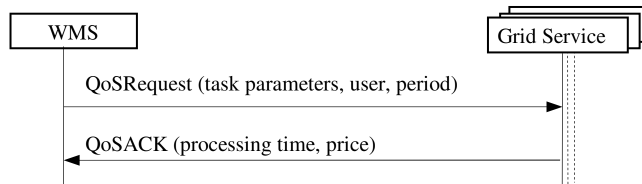


Fig. 4. QoS request scenario.

The knowledge of QoS details for all available services is the key to scheduling workflow tasks efficiently. A possible QoS request scenario is presented in Figure 4. It initially starts from the Workflow Management System (WMS) sending a QoS request to the Grid services for every task. In the request it indicates the task parameters, user of the workflow and the estimated execution period. On receiving the request, the Grid services reply with the QoS parameters (e.g. processing speed, available storage space and free memory) of the service they can offer and the corresponding price for delivering the service at the specified QoS level.

C. *Workflow Scheduling*

Workflow scheduling focuses on mapping and managing the execution of workflow tasks onto grid services. For the "pay-per-use" service Grid, the scheduling decision during workflow scheduling must be guided by users' QoS constraints. There are three major steps in workflow scheduling: workflow planning, advance reservation, workflow execution with run-time rescheduling.

1) Workflow Planning

Workflow planning is to select a service for every task in the workflow and generate a schedule before workflow execution. The result of the schedule must satisfy users' QoS constraints. The decision making of the planner for workflow execution needs to reference the entire workflow according to the QoS parameters of services obtained from QoS requests. In general, mapping tasks on distributed services is an NP-hard problem; the workflow planner may only produce a sub-optimal schedule in order to balance the scheduling time.

2) Advance Reservation

An advance reservation function has been proposed to be supported by guaranteed QoS services [1]. It is important to workflow scheduling especially for long lasting workflow execution. Workflow management systems need to make reservation of services selected by the planner in advance to ensure the availability of services.

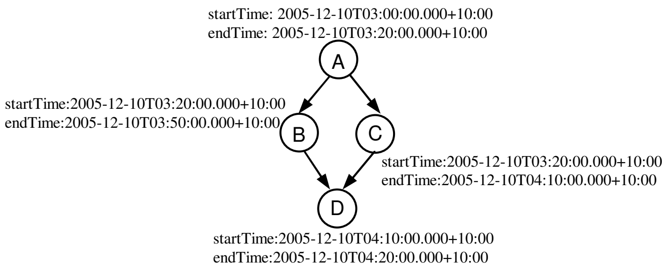


Fig. 5. Possible reservation schedule.

The time slots for advance reservation services can be generated based on every optimal service and possible start time of the workflow execution. Figure 5 illustrates a possible advance reservation schedule for workflow execution. The earliest start time of the task depends on the possible completion time of its parent tasks. If a task has more than one predecessor, the start time is the latest completion time of its predecessors. If we consider communication overhead, the task start time will be the latest completion time of parent tasks plus the communication time.

However, the time slots of desired services requested by the result of planning may not be available when the workflow system makes the reservations. Therefore, the workflow scheduling needs to be able to re-plan so that it can acquire an alterative schedule.

3) Workflow execution with run-time rescheduling

Typical utility computing services are QoS guaranteed and need to meet service commitments. However, there is still a possibility that services may violate the contract between the workflow system and service provider for reasons such as service failure and service delay due to the competition with other service consumers with higher priority. Therefore, the workflow scheduler must be able to adapt and update the schedule based on resource dynamics. For example, if a task execution is delayed behind the desired start time of its children tasks, the scheduling must adjust the reservation schedule for unexecuted tasks. A QoS monitor is required in the system to monitor the agreed performance and inform the planner of any changes.

For non-reservation services, service availability can only be known at run-time. In this case, run-time rescheduling is more critical. In addition to dealing with the situations of contract violation, rescheduling also needs to handle unavailability of optimal services at the time of a task execution.

4) Service Level Agreement

In the service Grid, the actual allocation of services is not under the control of workflow management system. The commitment for service execution is based on the Service Level Agreement (SLA) between the workflow management system and service providers. An SLA is a contract that specifies the minimum expectations and obligations that exist between consumers and providers [2]. SLA parameters for workflow tasks are QoS requirements of task processing and they include performance objectives such as earliest start time and latest completion time, and a rate model such as processing price.

Penalty clauses for service level violation are also required in an SLA to enforce service level guarantees. The penalty levels for service execution violation may vary for different workflow tasks. For example, if the service for executing task B in Figure 5 is delayed for 20 minutes, it does not affect the completion of the overall workflow. However, with any delay of executing task C, the whole workflow execution will delay. Therefore, the penalty levels for workflow task processing should be based on the degree of impact on the whole workflow execution rather than on a single service execution.

III. A QoS-based Workflow Scheduling

The processing time and execution cost are two typical QoS constraints for executing workflows on "pay-per-use" services. The users normally would like to get the execution done at lowest possible cost within their required timeframe. Given this motivation, in this section we present a QoS-based workflow scheduling methodology and algorithm that allows the workflow management system to minimize the execution cost while delivering results within the deadline.

1) Problem Description and Methodology

We model workflow applications as a Directed Acyclic Graph (DAG). Let Γ be the finite set of tasks T_i $(1 \le i \le n)$. Let Λ be the set of directed arcs of the form (T_i, T_j) where T_i is called a parent task of T_j, and T_j the child task of T_i. We assume that a child task cannot be executed until all of its parent tasks are completed. Let D be the time constraint (deadline) specified by the users for workflow execution. Then, the workflow application can be described as a tuple $\Omega(\Gamma, \Lambda, D)$.

In a workflow graph, we call a task which does not have any parent task an *entry task* denoted as T_{entry} and a task which does not have any child task an *exit task* denoted as T_{exit}.

Let m be the total number of services available. There are a set of services S_i^j:cond $\equiv (1 \le i \le n,\ 1 \le j \le m_i,\ m_i \le m)$ is capable of executing the task T_i, but only one service can be assigned for the execution of a task. Services have varied processing capability delivered at different prices. In general, the service price is inversely proportional to the processing time as shown in Figure 6. We denote t_i^j (such that cond is satisfied) as the sum of the processing time and data transmission time, and c_i^j (such that cond is satisfied) as the sum of the service price and data transmission cost for processing T_i on service S_i^j.

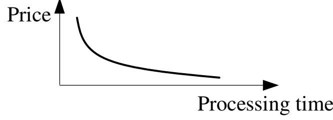


Fig. 6. Processing time vs. price for task execution.

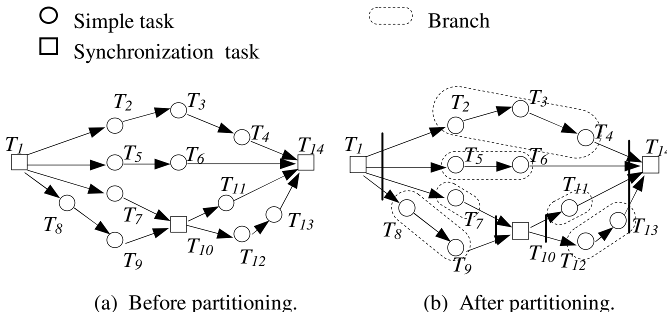
The scheduling problem is to map every T_i onto some S_i^j to achieve minimum execution cost and complete the workflow execution within the deadline D. We solve the scheduling problem by following the divide-and-conquer technique using the methodology listed below:

- **Step 1**. Discover available services and request QoS parameters of services for every task.
- **Step 2**. Group workflow tasks into task partitions.
- **Step 3**. Distribute user's overall deadline into every task partition.
- **Step 4**. Generate optimized schedule plan based on the local optimal solution of every task partition.
- **Step 5**. Start workflow execution and reschedule when the initial schedule is violated at run-time.

We provide details of steps 2-5 in the following sub-sections. The service discovery can be done by querying a directory service such as the Grid market directory [14] .

B. Workflow Task Partitioning

We categorize workflow tasks to be either a *synchronization task* or a *simple task*. A synchronization task is defined as a task which has more than one parent or child task. In Figure 7a, T_1, T_{10} and T_{14} are synchronization tasks. Other tasks which have only one parent task and child task are simple tasks. In the example, $T_2 - T_9$ and $T_{11} - T_{13}$ are simple tasks.



(a) Before partitioning. (b) After partitioning.

Fig. 7. Workflow task partition.

Let a *branch* be a set of simple tasks that are executed sequentially between two synchronization tasks. For example, the branches in Figure 7b are $\{T_2, T_3, T_4\}$, $\{T_5, T_6\}$, $\{T_7\}$, $\{T_8, T_9\}$, $\{T_{11}\}$ and $\{T_{12}, T_{13}\}$. We then partition workflow tasks Γ into independent branches $B_i (1 \le i \le k)$ and synchronization tasks $Y_i (1 \le i \le l)$, such that k and l are the total number of branches and synchronization tasks in the workflow respectively.

Let V be a set of nodes in a DAG corresponding to a set of task partitions $V_i (1 \le i \le k + l)$. Let E be the set of directed edges of the form (V_i, V_j) where V_i is a parent task partition of V_j and V_j is a child task partition of V_i. Then, a task partition graph is denoted as $G(V, E, D)$. A *simple path* (referred to as path) in G is a sequence of task partitions such that there is a directed edge from every task partition (in the path) to its child, where none of the vertices (task partitions) in the path is repeated.

A task partition V_i has four attributes: start time ($st[V_i]$), deadline ($dl[V_i]$), expected execution time ($eet[V_i]$), and minimum execution time ($met[V_i]$). The earliest start time of V_i is the earliest time the first task in it can be executed and it can be computed according to its parent partitions, $st[V_i] = \max_{V_j \in P_i} dl[V_j]$, where P_i is the set of parent task partitions of V_i. The minimum execution time of V_i is $\sum_{T_x \in V_i} \min_{1 \le y \le m_x} t_x^y$. The attributes are related as: $eet[V_i] = dl[V_i] - st[V_i]$.

C. Deadline Assignment

After workflow task partitioning, we distribute the overall deadline between each V_i in G. The deadline $dl[V_i]$ assigned to any V_i is a *sub-deadline* of the overall deadline D. In this paper, we consider the following deadline assignment policies:

P1. *The cumulative sub-deadline of any independent path between two synchronization tasks must be same.*

A synchronization task cannot be executed until all tasks in its parent task partitions are completed. Thus, instead of waiting for other independent paths to be completed, a path capable of being finished earlier can be executed on slower but cheaper services. For example, the deadline assigned to $\{T_8, T_9\}$ is the same as $\{T_7\}$ in Figure 7. Similarly, deadlines assigned to $\{T_2, T_3, T_4\}$, $\{T_5, T_6\}$, and $\{\{T_7\}, \{T_{10}\}, \{T_{12}, T_{13}\}\}$ are same.

P2. *The cumulative sub-deadline of any path from $V_i (T_{entry} \in V_i)$ to $V_j (T_{exit} \in V_j)$ is equal to the overall deadline D.*

P2 assures that once every task partition is computed within its assigned deadline, the whole workflow execution can satisfy the user's required deadline.

P3. *Any assigned sub-deadline must be greater than or equal to the minimum processing time of the corresponding task partition.*

If the assigned sub-deadline is less than the minimum processing time of a task partition, its expected execution time will exceed the capability that its execution services can handle.

P4. *The overall deadline is divided over task partitions in proportion to their minimum processing time.*

The execution times of tasks in workflows vary; some tasks may only need 20 minutes to be completed, and some others may need at least one hour. Thus, the deadline distribution for a task partition should be based on its execution time. Since there are multiple possible processing times for every task, we use the minimum processing time to distribute the deadline.

We implemented deadline assignment policies on the task partition graph by combining Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms with critical path analysis to compute start times, proportion and sub-deadlines of every task partition.

D. Planning

The planning stage is to generate an optimized schedule for advance reservation and run-time execution. The schedule allocates every workflow task to a selected service such that they can meet users' deadline at low execution cost.

We solve the workflow scheduling problem by dividing the entire problem into several task partition scheduling problems. Once each task partition has its own sub-deadline, we can find a local optimal schedule for each task partition. If each local schedule guarantees that their task execution can be completed within their sub-deadline, the whole workflow execution will be completed within the overall deadline. Similarly, the result of the cost minimization solution for each task partition leads to an optimized cost solution for the entire workflow. Therefore, an optimized workflow schedule can be easily constructed by all local optimal schedules.

There are two types of task partitions: synchronization task and branch partition. The scheduling solutions for each type of partition and the overall algorithm are described in following sub-sections.

1) *Synchronization Task Scheduling* (STS)

For STS, the scheduler only considers one task to decide the service for executing that task. The objective function for scheduling of a synchronization task Y_i is:

$$\min c_i^j \text{, where } 1 \le j \le m_i \text{ and } t_i^j \le eet(Y_i)$$

The solution to a single task scheduling problem is simple. The optimal decision is to select the cheapest service that can process the task within the assigned sub-deadline.

2) *Branch Task Scheduling* (BTS)

If there is only one simple task in a branch, the solution for BTS is the same as STS. However, if there are multiple tasks, the scheduler needs to make a decision on which service to execute its child task after the completion of the parent task. The optimal decision is to minimize the total execution cost of the branch and complete branch tasks within the assigned sub-deadline. The objective function for scheduling branch B_j is:

$$\min \sum_{T_i \in B_j} c_i^k \text{, where } 1 \le k \le m_i \text{ and } \sum_{T_i \in B_j} t_i^k \le eet(B_j)$$

BTS can be achieved by modeling the problem as a Markov Decision Process (MDP) [12], which has been shown to be effective for solving sequential decision problems.

3) *MDP Model for Sequential Branch Tasks*

The definition of our MDP model for scheduling branch B_i is described below:

States:

A Markov decision process is a state space S such that:

Definition 1: *A state $s \in S$ consists of current execution task T_i and remaining deadline RD.*

Definition 2: *A start state is a state when the current execution task is the first task of the branch and RD is $dl[B_i]$.*

Definition 3: *A terminal state is a state after the last task of the branch is completed.*

Actions and transitions:

For every state s, there are a set of actions A_s. Actions incur immediate utility and affect the MDP to transit from one state to another.

Definition 4: *An action in the MDP is to allocate a service to a task. There are two variables associated with each action a: the processing time of the service denoted as t and the service price denoted as c.*

Definition 5: *$u(s,a,s')$ is the immediate utility obtained from taking action a at state s and transitioning to state s'.*

$$u(s,a,s') = \begin{cases} \infty, & s'.RD < 0 \\ a.c, & \text{otherwise} \end{cases}$$

Definition 6: *A transition incurred by an action from one state to another is deterministic, as services are QoS guaranteed.*

The MDP problem is to find an optimal policy π^* for all possible states. A policy is a mapping from s to a. Decision making for finding an optimal action for each state is not based on the immediate utility of the action but its expected utility, which is the sum of all the immediate utilities obtained as a result of decisions made for transiting from this state to a terminal state.

The value associated to each state represents the expected utility of this state in the MDP. This value is calculated recursively by using the value of successor states. The value of one state s is:

$$U(s) = \min_{a \in A_s} \{u(s,a,s') + U(s')\}$$

The best action for state s is:

$$\pi^*(s) = \arg\min_{a \in A_s} \{u(s,a,s') + U(s')\}$$

The computation of the optimal policy can be solved by using a standard dynamic programming algorithm such as policy iteration and value iteration [12] (we have used value

iteration here). The optimal policy indicates the best services that should be assigned to execute branch tasks under a specific sub-deadline.

4) Planning Algorithm

Figure 8 shows the pseudo-code of the algorithm for planning an execution schedule. After acquiring the information about available services for each task, a task partition graph G is generated from the application graph Ω and overall deadline D is distributed over every partition in it. Then optimal schedules are computed for every partition in G level-by-level using either STS or BTS. We also found that after the optimization of one partition, there is an idle time between expected completion of planned services and assigned sub-deadline. Instead of waiting, we adjust the assigned sub-deadline of planned partitions and the start time of their child partitions.

```
Input: A workflow graph Ω(Γ,Λ,D)
Output: a schedule for all workflow tasks
1    request processing time and price from available services for ∀T_i ∈ Γ
2    convert Ω into G(V,E,D)
3    distribute deadline D over ∀V_i ∈ G
4    for all V_i ∈ G do    scheduled[i] ← false
5    S_1 ← all entry partitions
6    for all i ∈ S_1 do
7        if i is a branch then
8            compute an optimal schedule for i using BTS
9        else
10           compute an optimal schedule for i using STS
11       scheduled[i] ← true
12       Child-PartitionHandling(i)  [see below]
13   while Q is not empty do
14       i ← remove the first element in Q
15       S_2 ← all parent task partitions of i
16       if ∀ j ∈ S_2, scheduled[j] is true then  st[i] ← max_{j∈S3} dl[j]
17       else  put i into Q
18       compute an optimal schedule for i using STS
19       Child-PartitionHandling(i)

Child-PartitionHandling( i):
20   dl[i] ← get expected completion time of i
21   S_2 ← get child task partitions of i
22   for all j ∈ S_2 do
23       if j is a branch partition then
24           st[j] ← dl[i]
25           compute an optimal schedule for j using BTS
26           k ← get the child partition of j
27           put k into a queue Q
28       else  put j into Q
```

Fig. 8. Planning algorithm for optimizing execution cost within users' deadline. STS is to compute an optimal schedule for a synchronization task to optimize the execution within sub-deadline while BTS is for branch tasks.

E. Rescheduling

In order to complete workflows and satisfy users' requirements, run-time rescheduling is required to be able to adapt to dynamic situations such as the variation in availability of services due to failures. The key idea of our rescheduling policy for handling an unexpected situation is to adjust sub-deadlines and re-compute optimal schedules for unexecuted task partitions level-by-level. The motivation of the level-by-level task partition approach is to reschedule the minimum number of task partitions. For example, if the execution of one task partition is delayed, we look at its child task partitions. If the delay time can be accommodated by the child task partitions, rescheduling will not impact on its lower levels. Otherwise, the rest of the delay time is accumulated to its successors until the total delay time has been distributed.

```
Input: A task partition graph G(V,E,D), delayed synchronization task X
and delay time delay
Output: a new schedule for the unexecuted tasks in the workflow
1    S_1 ← all child task partitions of X
2    for all i ∈ G do  scheduled[i] ← true
4    for all i ∈ S_1 do  st[i] ← st[i] + delay
6    PartitionRescheduling(i)
7    while Q is not empty do
8        i ← remove the first task partition in Q
9        S_3 ← all parent task partitions of i
10       if ∀ j ∈ S_2, scheduled[j] is true then  st[i] ← max_{j∈S3} dl[j]
12       PartitionRescheduling( i )

PartitionRescheduling( i ):
13   if (dl[i] - st[i]) ≥ me[i] then
14       compute a new optimal schedule for i
15       scheduled[i] ← true
16   else  dl[i] ← st[i] + me[i]
17       compute a new optimal schedule for i
18       scheduled[i] ← true
19       S_4 ← all child task partitions of i
20       for all j ∈ S_4 do
21           put j into Q
22           scheduled[j] ← false
```

Fig. 9. Rescheduling algorithm for a synchronization delay.

The rescheduling algorithm for a synchronization task delay is illustrated in Figure 9. First, we adjust the start time of child task partitions to be the actual completion time of the delayed synchronization task (line 4). Then, we check whether the new deadlines of the child task partitions can be achieved by comparing their minimum processing times (line 13). If achievable, the planner generates new optimal schedules for the tasks in the child task partitions based on the new expected execution times (line 14) and rescheduling is stopped. Otherwise, new sub-deadlines are assigned by using the minimum processing time as the expected execution time and then new schedules are generated (line 16-18). When the delay cannot be accommodated by the first level child partitions, the lower level child partitions are put into the queue for further rescheduling (line 19-22). A queue, Q, is used for implementing the breadth-first search algorithm for identifying new start time in the graph.

For branch task rescheduling, if a branch task execution is delayed, the optimal schedule for the next branch task of the delayed task can still be obtained from the initial MDP result, according to its current remaining sub-deadline. The other

unexecuted partitions will not be affected as long as the delay does not exceed the minimum processing time of the remaining unexecuted tasks in the branch.

In addition to handling task execution delay, the level-by-level task partition based approach can also be applied for managing other dynamic situations such as service unavailability and service policy change.

IV. Performance Evaluation

The performance of QoS-based workflow scheduling algorithm described in Section III has been evaluated through simulation using the GridSim Toolkit [4]. We conducted several experiments by simulating the structure of a protein annotation workflow application (see Figure 10) developed by the London e-Science Centre [3]. The number in bracket next to the task represents the length of task in MI (million instructions). Every task in the workflow requires a certain type of service for processing.

We simulated 15 types of services and each service type is supported by 5 different service providers. That is, we simulated 80 service providers. Table I shows attributes an instance of 5 different service providers in terms of their processing capacity in MIPS (Million Instructions Per Second), delivery/processing time in second and price in G$. They all deliver the same type of service required for executing task 3. We extended GridSim to support service discovery with request based on QoS parameters. As indicated in Figure 11, the workflow system first discovers available services for every task via Grid Index Service (GIS) within GridSim and then queries the services to obtain their processing time and price. The processing time of a task on a service depends on the complexity of the task and the combined capability of resource used for service provision. As indicated in Figure 6, services with lower processing time are delivered at higher price.

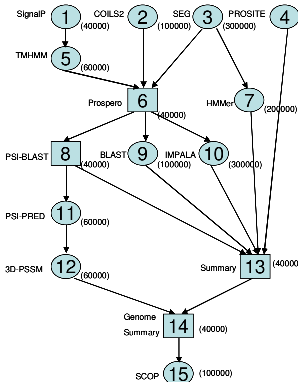


Fig. 10. A workflow for the protein annotation.

Table I. QoS attributes of services of different providers for executing task 3.

Service ID	MIPS Rating	Processing Time (sec)	Cost (G$)
1	500	60	6
2	1000	30	12
3	1500	20	18
4	2000	15	24
5	2500	12	30

In our first experiment, we compare our proposed scheduling algorithm denoted as *Deadline Min-Cost* with three other scheduling algorithms: *Greedy-Cost*, *Greedy-Time* and *100-Random Selection*. The Greedy-Cost and Greedy-Time algorithms always arrive at the best immediate solution while searching for an answer. The Greedy-Cost algorithm selects the cheapest service for executing each task, whereas the Greedy-Time algorithm selects the fastest service. The 100-Random Selection algorithm uses the average value of execution time and execution cost captured through the repeated execution of workflow application 100 times in which services are selected randomly.

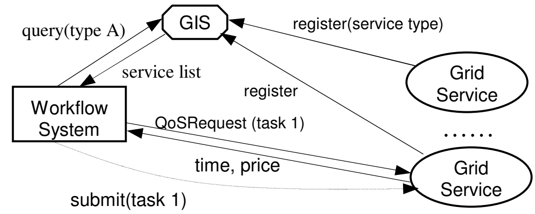
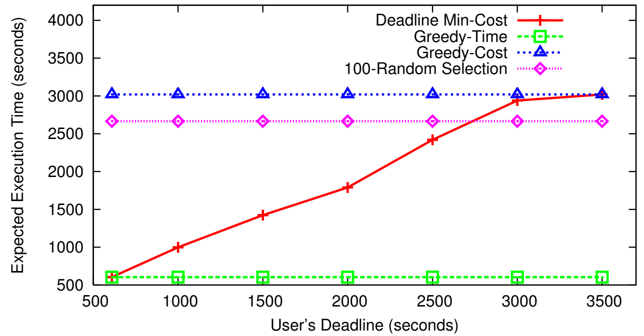


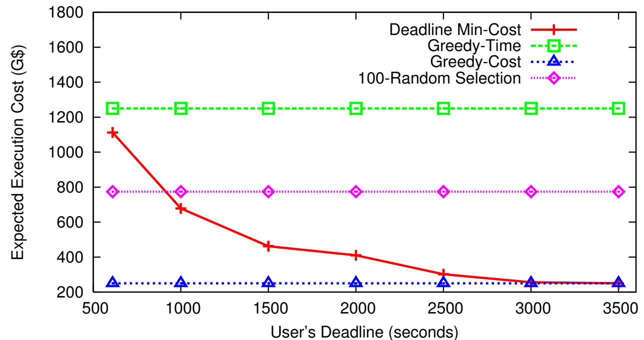
Fig. 11. Service discovery in GridSim.

The two main measurements used to evaluate the scheduling approaches are the time constraint and execution cost. The former indicates whether the schedule produced by the scheduling approach meets the required deadline, while the latter indicates how much it costs to schedule the workflow tasks on the simulated service Grid.

Figure 12 compares the execution time and cost generated by the planner using the four scheduling approaches. As shown in Figure 12a, the expected execution time of workflow using Deadline Min-Cost algorithm increases as users relax their deadline. The workflow execution time for Greedy-Cost and 100-Random Selection algorithms is higher and cannot meet the users' requirements when the deadline is lower. The Greedy-Time algorithm can complete earlier than the Deadline Min-Cost algorithm but its execution cost is much higher (Figure 12b). Figure 12b shows that the execution cost of workflow using Deadline Min-Cost algorithm is reduced as users relax their deadline.



a. Expected execution time of four scheduling approaches.



b. Expected execution cost of four scheduling approaches.

Fig. 12. Expected execution time and cost using four scheduling approaches. Greedy-Time can complete the execution with earliest time, but the corresponding cost is very high. Greedy-Cost can complete the execution with cheapest cost, but it is unable to meet users' deadlines when the deadline is small.

We can see from Figure 12 that the Deadline Min-Cost algorithm is the only one that considers users' deadline requirements while optimizing the cost. The Greedy-Time and Greedy-Cost algorithms represent the scheduling approaches that intend to achieve minimization of execution time and cost respectively.

In another experiment, we executed the workflow with the optimal services produced by the planner using the Deadline Min-Cost algorithm. At run-time, we simulated delays for the execution of task 6 as 0, 50, 100, 150, 200, 250 and 300 seconds. Figure 13 shows that actual workflow completion time with and without rescheduling. We can see that rescheduling is able to adapt to the delay time and complete the workflow execution on time. However, the actual execution cost increases (Figure 14), since the scheduler switches the remaining tasks to more expensive services to speed up execution. Therefore, there is a need for appropriate penalty mechanisms to compensate for the loss caused by the violation of the QoS guarantees.

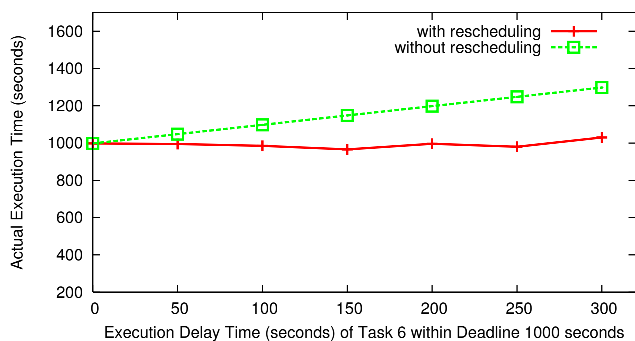


Fig. 13. Actual execution time of task 6 (deadline 1000 seconds) with rescheduling and without rescheduling for increasing delay.

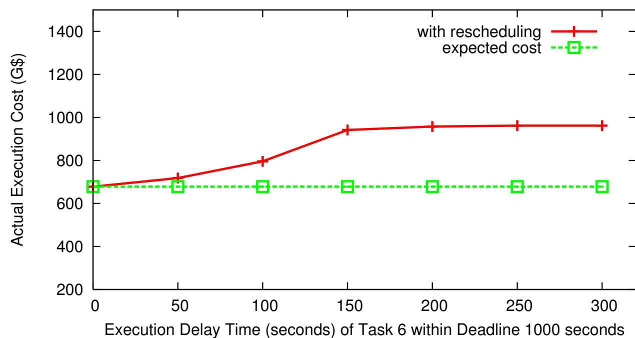


Fig. 14. Actual execution cost of task 6 (deadline 1000 seconds) with rescheduling for increasing delay.

VI. Conclusion and Future Work

Workflow management on "pay-per-use" service Grids has not been addressed in existing Grid workflow systems. In this paper, we presented a QoS-based workflow management system. In this, we proposed a novel QoS-based workflow scheduling algorithm that minimizes the cost of execution while meeting the deadline. We also described task partitioning and overall deadline assignment for optimized execution planning and efficient run-time rescheduling. We have utilized a Markov Decision Process approach to schedule sequential workflow task execution.

The current system uses run-time rescheduling to handle service agreement violations. In future work, we will further enhance our scheduling method to handle more dynamic scenarios such as dynamic pricing.

Acknowledgments

We would like to thank Hussein Gibbins, Chee Shin Yeo, Srikumar Venugopal, Sushant Goel, and Arun Konagurthu for their comments on this paper. This work is partially supported through StorageTek Fellowship and Australian Research Council (ARC) Discovery Project grant.

References

[1] S. Benkner, I. Brandic, G. Engelbrecht, R. Schmidt, "VGE - A Service-Oriented Grid Environment for On-Demand Supercomputing", In *the Fifth IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, Pittsburgh, PA, USA, November 2004.

[2] M.J. Buco et al, "Utility computing SLA management based upon business objectives," IBM System Journal, Vol. 43(1):159-178, 2004.

[3] A. O'Brien, S. Newhouse and J. Darlington, "Mapping of Scientific Workflow within the e-Protein project to Distributed Resources", In *UK e-Science All Hands Meeting*, Nottingham, UK, Sep. 2004.

[4] 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,* Vol. 14(13-15):1175-1220, Wiley Press, USA, 2002.

[5] K. Cooper et al, "New Grid Scheduling and Rescheduling Methods in the GrADS Project", *NSF Next Generation Software Workshop*, International Parallel and Distributed Processing Symposium, Santa Fe, IEEE CS Press, Los Alamitos, CA, USA, April 2004.

[6] E. Deelman et al, "Mapping Abstract Complex Workflows onto Grid Environments", *Journal of Grid Computing*, Vol.1:25-39, 2003.

[7] T. Eilam et al, "A utility computing framework to develop utility systems", *IBM System Journal*, Vol. 43(1):97-120, 2004.

[8] T. Fahringer et al, "ASKALON: a tool set for cluster and Grid computing", *Concurrency and Computation: Practice and Experience*, 17:143-169, Wiley InterScience, 2005.

[9] I. Foster et al, "The Physiology of the Grid", Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[10] A. Mayer et al, "ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time", In *UK e-Science All Hands Meeting*, Nottingham, UK, IOP Publishing Ltd, Bristol, UK, September 2003.

[11] S. Newhouse, "Grid Economy Services Architecture (GESA)", Grid Economic Services Architecture WG, Global Grid Forum, 2003.

[12] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction", MIT Press, Cambridge, MA, 1998.

[13] G. Thickins, "Utility Computing: The Next New IT Model", *Darwin Magazine*, April 2003.

[14] J. Yu, S. Venugopal, and R. Buyya, "A Market-Oriented Grid Directory Service for Publication and Discovery of Grid Service Providers and their Services", *Journal of Supercomputing*, Kluwer Academic Publishers, USA, 2005.

[15] J. Yu and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing", Technical Report, GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, March 10, 2005.

[16] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor - A Distributed Job Scheduler. *Beowulf Cluster Computing with Linux*, The MIT Press, MA, USA, 2002.

[17] G. Singh, E. Deelman, G. Mehta, K. Vahi, M. Su, B. Berriman, J. Good, J. Jacob, D. Katz, A. Lazzarini, K. Blackburn, S. Koranda, "The Pegasus Portal: Web Based Grid Computing", *The 20th Annual ACM Symposium on Applied Computing*, Santa Fe, NM, Mar. 13 -17, 2005.

[18] A. Birnbaum, J. Hayes, W. Li, M. Miller, P. Bourne, H. Casanova, "Grid workflow software for High-Throughput Proteome Annotation Pipeline", *Proceedings of the First International Workshop on Life Science Grid (LSGRID2004)*, Ishikawa, Japan, June 2004.