Computing and Informatics, Vol. 37, 2018, 838–864, doi: 10.4149/cai_2018_4_838

COST-EFFICIENT SCHEDULING FOR DEADLINE CONSTRAINED GRID WORKFLOWS

Alireza Dehlaghi-Ghadim

School of Electrical and Computer Engineering
University of Tehran
Tehran, Iran
e-mail: a.dehlaghi@ut.ac.ir

Reza Entezari-Maleki

School of Computer Science
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
e-mail: entezari@ipm.ir

Ali Movaghar

Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
e-mail: movaghar@sharif.edu

Abstract. Cost optimization for workflow scheduling while meeting deadline is one of the fundamental problems in utility computing. In this paper, a two-phase cost-efficient scheduling algorithm called *critical chain* is presented. The proposed algorithm uses the concept of slack time in both phases. The first phase is deadline distribution over all tasks existing in the workflow which is done considering critical path properties of workflow graphs. *Critical chain* uses slack time to iteratively select most critical sequence of tasks and then assigns sub-deadlines to those tasks. In the second phase named mapping step, it tries to allocate a server to each task considering task's sub-deadline. In the mapping step, slack time priority in select-

ing ready task is used to reduce deadline violation. Furthermore, the algorithm tries to locally optimize the computation and communication costs of sequential tasks exploiting dynamic programming. After proposing the scheduling algorithm, three measures for the superiority of a scheduling algorithm are introduced, and the proposed algorithm is compared with other existing algorithms considering the measures. Results obtained from simulating various systems show that the proposed algorithm outperforms four well-known existing workflow scheduling algorithms.

Keywords: Grid computing, workflow, slack time, critical path, cost-based scheduling

Notations

T	Set of all tasks in the application
E	Set of all dependencies in the application
t_i	Task i
e_{ij}	Dependency between task i and task j
δ	Deadline of the application
S	Set of servers
s_i	Server i
T_{ij}	Processing time of task t_i on server s_j
C_{ij}	Processing cost of task t_i on server s_j
$S(t_i)$	The server allocated to execute task t_i
$T_{iS(t_i)}$	Execution time of task t_i on $S(t_i)$
$\text{Delay}(t_i, t_j)$	Data transmission time between task t_i and task t_j on the link between $S(t_i)$ and $S(t_j)$
$imm\_preds(t_i)$	All tasks in the workflow graph in which t_i is their immediate successor
$imm\_succs(t_i)$	All tasks in the workflow graph in which t_i is their immediate predecessor
$\text{MET}(t_i)$	Minimum time for the execution of task t_i on the fastest server
$\text{MTT}(e_{ij})$	Minimum data transmission time between task t_i and task t_j
$\text{EST}(t_i)$	Earliest start time of task t_i
$\text{LFT}(t_i)$	Latest finish time of task t_i
$\text{ST}(t_i)$	Slack time of task t_i
CST	Chain start time
CFT	Chain finish time
SST	Schedule start time
SFT	Schedule finish time
NC	Normalized cost
θ	Deadline factor
T_{min}	Minimum execution time of the application

1 INTRODUCTION

Grid is an infrastructure with the aim of solving high scale problems in science, economy, aerology, engineering and many other fields [1]. Resource sharing is one of the most significant advantages of grid computing [2]. The most important resources shared in grids include CPU, main memory, secondary memory, network bandwidth, and data. Traditional resource management techniques provide no incentive for users to share resources fairly. Consequently, to support different levels of Quality of Service (QoS) and manage priority between user applications, utility grid computing has been emerged [3, 4]. In this paradigm, users have to pay for each time they use servers with specific QoS. How to allocate grid servers to the tasks to satisfy the specific needs is one of the important challenges in this area.

This paper focuses on workflow scheduling with the aid of heuristics. In this case, workflows are composed of several tasks with partial order, in the way that some tasks have control or data dependencies on the others. Many complex applications in different domains such as e-science as bioinformatics and astronomy, and e-business can be modeled as workflows [5]. To solve the applications, the resulted workflows need to be processed, so the tasks should be executed based on their dependencies [6]. We can describe workflows with Directed Acyclic Graphs (DAGs) in which each node in DAG represents a specific task in the corresponding workflow. Therefore, the scheduling problem can be stated as assigning a DAG of tasks to the limited processing units according to their requirements and transposition constraints. To solve this type of scheduling problems, two different approaches can be used: *approximation* and *heuristic*. In the approximate algorithms, since it is unlikely that there can ever be efficient polynomial-time exact algorithms solving NP-hard problems, one settles for polynomial-time sub-optimal solutions so called approximation, which uses formal computational models to obtain sufficiently good solution instead of searching the entire solution space for an optimal solution. Heuristic represents the class of algorithms which makes more realistic assumptions about a priori knowledge concerning process and system loading characteristics [7, 8].

Generally, mapping tasks on distributed servers is an NP-hard problem, and workflow scheduling as an optimization problem produces large scheduling overhead, especially for problems with two-dimensional constraints such as time and cost [9, 10]. The most well-known goal considered for workflow scheduling is minimizing the makespan of the application. Although many research papers have addressed this problem [9, 11], in the economic scheduling, cost reduction along with satisfying the deadline is very important and that should be taken into consideration in workflow scheduling [12]. Consequently, traditional approaches for scheduling tasks in grid community are no longer suitable for utility grids. Therefore, some new methods have been proposed in the past years to fulfill this requirement [4, 6, 9, 10, 13, 14, 15, 16, 17, 30]. Many recent approaches in workflow scheduling consider critical path as a hint to assign sub-deadlines to the tasks [11, 13, 17, 18, 19], but deadline distribution with those methods is not efficient enough to decrease deadline violations. Another disadvantage of the previously presented scheduling methods is

the lack of priority between tasks in the mapping step. To overcome these shortcomings, we develop a new scheduling algorithm with two steps. In the first step, an efficient method for initial distribution of tasks deadlines is presented, and in the second step, scheduling priority for the task with minimum slack time is considered to reach a better result. In order to evaluate the proposed algorithm and compare it with others, we simulate three types of well-known workflows under various assumptions and system configurations. Simulation results show the advantage of the proposed algorithm in comparison with four most-cited recent algorithms.

The remainder of this paper is organized as follows. In Section 2, some related work done on scheduling problem, especially on workflow scheduling in grid environments, is presented. In Section 3, the scheduling problem in general case and its details in our context are described. The main proposed *critical chain* algorithm together with other sub-methods is presented in Section 4. In Section 5, experimental results obtained from simulation are given. Finally, Section 6 concludes the paper and presents the future work which can be done in this research field.

2 RELATED WORK

There are several research works addressing the problem of mapping workflows on multiprocessors [18, 20, 21]. However, some constraints like communication delays and specifically budget issues on economic grids make the previously done research work on multiprocessor systems inefficient when they are applied to the grids.

Foster et al. [22] have described a General-purpose Architecture for Reservation and Allocation (GARA) that supports QoS specification. Dogan et al. [23] have studied the scheduling of a set of independent tasks considering some QoS factors such as reliability, security and timeliness. Tabbaa et al. [24] have presented a new scheduling algorithm for DAG applications in clusters. The algorithm considers the failure of resources and tries to schedule tasks to the cluster servers to tolerate the faults occurred in the system. Entezari-Maleki et al. [25] have proposed a genetic-based task scheduling algorithm to minimize the makespan of grid applications. The algorithm proposed in [25] only considers the makespan as a QoS factor. However, there are few papers considering the cost of scheduling as a QoS factor. Kardani-Moghaddam et al. [26] have proposed a hybrid genetic algorithm and variable neighborhood search which uses a fitness function to balance between makespan and execution cost of each scheduling solution. Agrawal et al. [27] have explored linear workflow scheduling for linear workflows, and found an approximation algorithm to maximize throughput in the one-port model. Moreover, they proved that finding a schedule respecting a given period and a given latency is NP-hard.

Yu et al. [9] have proposed the deadline-MDP algorithm that divides a DAG to partitions and then distributes the deadline over the partitions. Finally, deadline-MDP algorithm tries to locally optimize the cost for each partition using Markov models. It has been shown that deadline-MDP algorithm outperforms previous methods such as DTL and greedy cost [9, 10]. The genetic algorithm was used to

optimize the time of scheduling under budget constraint in [6]. Zhao et al. [28] have proposed two algorithms to schedule workflows with budget constraints. The first algorithm initially schedules all tasks to faster servers and then reschedules some tasks to meet the desires. Similarly, the second algorithm assigns each task to its cheapest server, and reschedules the tasks to the faster and more expensive servers as long as the budget is acceptable. According to Yuan et al. [29], Deadline Bottom Level (DBL) is a simple and efficient heuristics for workflow scheduling. In this method, all tasks are divided into several groups based on their bottom level with a backward method. The overall deadline is distributed over the groups considering maximum processing cycle of tasks in that group. All tasks in a group inherit a unique deadline of the corresponding group. Unlike the DTL method [9], the start time of each task is determined by the latest finish time of its immediate predecessors instead of the finish time of the last task in the previous level. Although DBL and DTL are effective and efficient, these algorithms show poor performance in firm deadlines. Yuan et al. [10] have presented the Deadline Early Tree (DET) method. First, they create *Early Tree* which is a spanning tree for primary schedule. Then they exploit dynamic programming to assign time window to each critical task, and consequently, find time window for non-critical tasks. Finally, the method tries to assign cheaper servers to each task according to its time window. The number of servers was assumed to be unlimited which is unrealistic assumption in most cases.

Cost-effective scheduling of deadline-constrained applications have been also investigated in hybrid clouds [15, 30, 31, 32, 33, 34]. Henzinger et al. [15] have designed a framework to handle cost-time trade-off in economic workflow scheduling called FlexPRICE. They tried to present the cost-time curve to enable users to select the appropriate deadline with an acceptable price. In fact, FlexPRICE was presented to solve cloud workflow programming, but the type of the problem is similar to the grid computing. Fard et al. [35] have introduced a pricing model and a truthful mechanism for scheduling single tasks considering monetary cost and completion time. With respect to the social cost of the mechanism, they extended the mechanism for dynamic scheduling of scientific workflows. Calheiros et al. [31] have presented an architecture for coordinated dynamic provisioning and scheduling which is able to cost-effectively complete applications within their deadlines. They considered the whole organization workload at individual tasks level, and their accounting mechanism was used to determine the share of the cost of utilization of public cloud resources to be assigned to each user. Poola et al. [30] considered deadline and budget constraints as QoS demanded by users, and tried to design a robust algorithm for scheduling of scientific workflows.

Abrishami et al. [13] have proposed a partial critical path scheduling based on properties of critical path. In deadline assignment step, the deadline is distributed over tasks, and then the cost of each allocation is locally optimized to provide the best possible result in each allocation. For deadline distribution, the method iteratively selects a sequence of tasks in the DAG and assigns the deadline to each member of that sequence. For this assignment, authors apply three different policies on the deadline distribution method: optimized policy, decrease cost policy, and fair

policy. The optimized policy iteratively tests all feasible assignments and selects the best one. It is obvious that this approach is time consuming, and it is not feasible for large-scale problems. The decrease policy is based on a greedy method which tries to approximate the optimized policy. In this policy, each task is assigned to the fastest server, and it is tried to decrease the cost by rescheduling a task to a cheaper server. The fair policy is similar to the decrease policy except that starting from the first task towards the last task in path, it substitutes the assigned server with the next slower server without exceeding sub-deadline. This procedure continues iteratively until no substitution can be made. According to the results reported in [13], the proposed algorithms show high performance in absence of server limitation.

3 PROBLEM DEFINITION

Directed Acyclic Graph (DAG) is one of the most acceptable models to represent workflow applications. Let $G(T, E)$ denote a DAG representing an application where T is the task set $T = \{t_1, t_2, \ldots, t_n\}$ in which n is the number of all tasks in the application. Moreover, edge set E represents the edges of the related DAG and shows the control or data dependencies between the tasks. The notation e_{ij} denotes an edge from the vertex t_i to t_j, and means that the task corresponding to the vertex t_j requires input data or command produced by execution of task t_i. Suppose that all tasks are topologically numbered in which each arc demonstrates the priority of $i < j$, means that execution of task t_j only depends on the tasks with lower numbers. The tasks having no input (output) edges are named *entry* (*exit*) tasks. For simplicity and without loss of generality, we suppose that always there is only one entry task in the application. If an application has more than one entry task, we can simply add a *dummy task* (a task that requires no processing) to it to produce our DAG of interest. Similarly, we can do the same for exit task in the graph. The number attached to each node represents the processing cycle of the corresponding task in the form of Million Instructions (MI). Also, the number attached to each arc e_{ij} shows the amount of data which should be sent from t_i to t_j. Figure 1 shows an example of DAG representation. In the graph represented in Figure 1, a node with index of i shows task t_i.

A service model in the utility grid computing consists of Grid Service Providers (GSPs) in which each of them provides some servers with specific QoS. The cost of processing in each server is proportional to the speed of process which means that if the scheduler allocates a faster server to execute a task, the user has to pay more cost [9]. Each server supports limited number of task types. We consider each of GSPs as a grid node. Assume that there are m service providers represented by set S where $S = \{s_1, s_2, \ldots, s_m\}$. Hence, for each task, there are several candidate servers which can execute the task. Assume T_{ij} is the processing time of task t_i executed on server s_j, and C_{ij} is its corresponding processing cost. If s_j is not capable of processing t_i, we consider both T_{ij} and C_{ij} to simply being infinity. It is assumed that dummy tasks can be processed on any server.

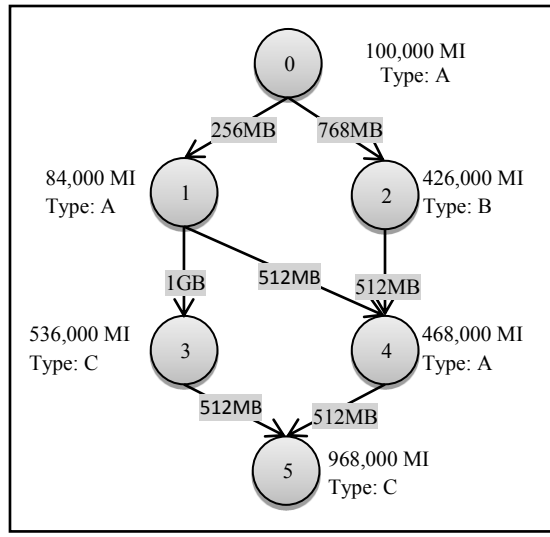

Figure 1. DAG representation of a sample workflow

In this paper, two well-known QoS measures in grids, execution time and cost, are considered. Therefore, our objective in this paper is to assign an appropriate server to the tasks to execute them with the goal of minimizing the overall execution cost while both tasks' precedence and application deadline are taken into account. To achieve this, we can consider workflow scheduling as an optimization problem with trade-off between time and cost [36]. Let δ denote a given deadline showing the latest possible finish time of the application or exit task. Let st_i and f_i denote the start and finish times of task t_i, respectively. Therefore, the workflow scheduling problem can be formulated as Equation (1).

$$\min \Sigma_{i\in T}\Sigma_{1\le k\le m} C_{ik}x_{ik}$$

$$S.t. \begin{cases} \Sigma_{1\le k\le m} x_{ik} = 1, & i \in T, \\ f_i < \delta, & i \in T, \\ f_i - st_i = T_{iS(t_i)}, & i \in T, \\ st_i > f_j + \text{Delay}(t_j, t_i), & j \in imm\_preds(i), \\ x_{ik} \in \{0,1\}, & 1 \le i \le n, 1 \le k \le m, \\ S(t_i) = S(t_j) \Rightarrow (st_i > f_j) \vee (f_i < st_j), & i,j \in T \end{cases} \tag{1}$$

where

$$x_{ij} = \begin{cases} 1, & t_i \text{ is assigned to } s_j, \\ 0, & \text{otherwise.} \end{cases}$$

The constraint $\Sigma_{1 \le k \le m} x_{ik} = 1$ in Equation (1) checks to make sure that there is a unique executor for each task. Similarly, condition $f_i < \delta$ ensures meeting the overall deadline. Moreover, the constraint $f_i - st_i = T_{iS(t_i)}$ checks the feasibility of task execution on the server in a given time slice, where $S(t_i)$ is the server assigned to execute task t_i, and $T_{iS(t_i)}$ is execution time of task t_i on $S(t_i)$. Each task would be executed on a resource only if its required data is transferred to the resource. This constraint is checked by $st_i > f_j + \text{Delay}(t_j, t_i)$, where $\text{Delay}(t_j, t_i)$ is data transmission time on the link between $S(t_i)$ and $S(t_j)$ which is computed as Equation (2).

$$\text{Delay}(t_i, t_j) = \frac{e_{ij}}{\text{bandwidth}(S(t_i), S(t_j))} \tag{2}$$

where $\text{bandwidth}(S(t_i), S(t_j))$ denotes the bandwidth of the link between servers executing tasks t_i and t_j.

In our model, it is considered that the number of servers is limited and some of the servers are busy in some cases, so they cannot be allocated to the tasks. The constraint $S(t_i) = S(t_j) \Rightarrow (st_i > f_j) \vee (f_i < st_j)$ checks if the same server is allocated to execute both tasks t_i and t_j. If it is, the start time of one of the tasks (e.g., task t_j) has to be greater than the finish time of the another one (e.g., task t_i). De et al. [36] showed that the time-cost optimization problem for DAG scheduling is a Discrete Time-Cost Trade-off Problem (DTCTP). DTCTP is an NP-hard problem, and the best-known solutions use dynamic programming, and branch and bound method to solve the problem. Unfortunately, these solutions are extremely time-consuming when the number of tasks and/or servers gets larger.

4 THE PROPOSED ALGORITHM

In order to efficiently schedule the tasks on the servers, we need an initial estimation of execution times of the tasks on servers. This estimation could help us to identify critical tasks of the application and schedule them on fast servers to meet the overall deadline of the application. So, in the first step, we propose an algorithm to divide the deadline on all tasks. After applying the algorithm of the first step, each task will have its own deadline in which the task should be processed before that deadline. The algorithm of the first step works based on the concept of *slack time* that is well-known in multiprocessor scheduling community. We use slack time in the deadline distribution algorithm to obtain the critical path not only for the whole graph, but also for all its sub-graphs. This method helps us to have fair deadline distribution. In the second step, in order to meet deadline as much as possible, critical tasks should be scheduled on faster servers. So, it is a good idea to have task priority in mapping step. Therefore, some models of priority are used in the mapping step and

then dynamic programming technique is exploited to have an efficient sequential task scheduling.

After describing the totality of the proposed algorithm, details of the algorithm are provided in the following sections. Some notations used in the algorithm are introduced in Section 4.1. Deadline distribution over all the tasks is described in Section 4.2. In Section 4.3, an illustrative example of the proposed deadline distribution algorithm is provided. After deadline distribution phase, partitioning technique is explained in Section 4.4. This technique is used to have better mapping for pipeline branches. Finally, in Section 4.5, the mapping algorithm based on priority method is explained.

4.1 Basic Definitions

Minimum Execution Time (MET) for task t_i refers to the minimum time for an execution of task t_i on the fastest available server which is capable of processing task t_i. Equation (3) shows MET calculation.

$$\mathrm{MET}(t_i) = \min_{s \in S} ET(t_i, s) \tag{3}$$

where S and $ET(t_i, s)$ denote the set of servers in the system and the execution time of task t_i on server s, respectively. In fact, $\min_{s \in S} ET(t_i, s)$ is the execution time of task t_i on the fastest available resource for executing t_i.

Minimum Transfer Time (MTT) for arc e_{ij} denoted by $\mathrm{MTT}(e_{ij})$ refers to the minimum time for data transmission of e_{ij} on the maximum available bandwidth of the grid. The calculations of $\mathrm{MTT}(e_{ij})$ is shown in Equation (4).

$$\mathrm{MTT}(e_{ij}) = \min_{\forall S(t_i), S(t_j) \in S} \mathrm{Delay}(t_i, t_j). \tag{4}$$

Earliest Start Time (EST) for each task refers to the earliest possible time that the task can start its execution. In other words, EST shows earliest possible start time of task t_i if all predecessors of t_i are executed on the fastest server(s). Similarly, Latest Finish Time (LFT) of a task refers to the latest possible finish time of the task while executing all successors of that task on the fastest server does not cause absolute deadline violation. Therefore, if execution of task t_i terminates at the time $\mathrm{LFT} + \epsilon$ and then all other tasks are executed on the fastest server, the execution of entire application will be finished on the time $\delta + \epsilon$. We can compute EST and LFT as Equation (5) and Equation (6), respectively.

$$\begin{aligned} \mathrm{EST}(t_{entry}) &= 0, \\ \mathrm{EST}(t_i) &= \max_{t_p \in imm\_preds(t_i)} \big(\mathrm{EST}(t_p) + \mathrm{MET}(t_p) + \mathrm{MTT}(e_{pi})\big), \end{aligned} \tag{5}$$

$$\begin{aligned} \mathrm{LFT}(t_{exit}) &= \delta, \\ \mathrm{LFT}(t_i) &= \min_{t_c \in imm\_succs(t_i)} \big(\mathrm{LFT}(t_c) - \mathrm{MET}(t_c) - \mathrm{MTT}(e_{ic})\big). \end{aligned} \tag{6}$$

Assume all predecessors and successors of task t_i are planned to be executed on the fastest server(s), so we have a wide range of time to schedule t_i. In other words, we are free to schedule task t_i in each part of this time period. Subtracting minimum execution time of task t_i from this time period, slack time of t_i shown by $\mathrm{ST}(t_i)$, is obtained. In fact, $\mathrm{ST}(t_i)$ is the maximum possible extra time for executing task t_i minus its minimum execution time. Equation (7) shows the slack time of task t_i.

$$\mathrm{ST}(t_i) = \mathrm{LFT}(t_i) - \mathrm{EST}(t_i) - \mathrm{MET}(t_i). \tag{7}$$

4.2 Deadline Distribution

Assigning a sub-deadline to each task according to the overall deadline of an application is the main objective of this step. In other words, we wish to assign a time window to each task for execution. This time window is determined by Schedule Start Time (SST) and Schedule Finish Time (SFT). During the deadline assignment, we are dealing with two types of tasks: *assigned* and *unassigned* tasks. If we assign a sub-deadline to a task (specifying both the start and finish times for the task), it is flagged as an assigned task, otherwise, it is called unassigned task. Dedicating the time interval of SST and SFT to the tasks does not mean that the tasks should be executed in these intervals; but these intervals give us an offline approximation of execution time of tasks based on other tasks within the workflow. The final goal of deadline distribution phase is assigning fair SST and SFT to each task. For this purpose, first, we have to compute earliest start time (EST) and latest finish time (LFT) of each task. Note that, for the assigned tasks, we do not compute EST and LFT measures, because they are equal to SST and SFT, respectively. Deadline assignment procedure is represented in Algorithm 1.

This algorithm begins with computing MET and MTT for all tasks, and then iteratively updates EST, LFT, and ST for each task and chooses a chain of unassigned tasks having minimum slack time. Considering this procedure, we certainly have a sequence of consecutive unassigned tasks that have minimum slack time. The first element of this sequence has assigned parent tasks and the last element has assigned child tasks. We call this sequence as *critical chain*. In other words, a *critical chain* is a sequence of unassigned tasks in which each task t_i is the immediate successor of task t_{i-1} in the chain. Moreover, all tasks in the *critical chain* have the same slack time which is the minimum one amongst slack times of all unassigned tasks within the application. If there is more than one *critical chain* with the aforementioned characteristic, one of them is arbitrarily chosen.

After selecting a *critical chain*, a time window can be assigned to each task of the chain. Let CST (Chain Start Time) denote a maximum LFT of immediate predecessors of the first task in the chain, and similarly, CFT (Chain Finish Time) denote a LFT of the last task in the chain. Now, we have a chain interval time CFT–CST which should be distributed over chain's tasks. We have already distributed this time interval to all tasks of the chain considering MET of each task. Actually, SST of the first task in the chain is its CST. Therefore, to obtain SFT

Algorithm 1: Deadline Distribution

```
1 Input: Application
  /* initialization                                                    */
  forall t_i ∈ Application do
      Compute MET(t_i);
      Compute MTT(t_i);
  end
  while (there are unassigned tasks) do
      Update EST and LFT for all unassigned tasks;
      minSlack ⟵ ∞;
      chain ← null;
      /* Check for critical sequence of task to schedule               */
      for i = 0 to i = numberOfTasks − 1 do
          if IsAssigned(t_i) then
              continue;
          end
          ST(t_i) ⟵ LFT(t_i) − EST(t_i) − MET(t_i) ;
          if ST(t_i) < minSlack then
              sequence ⟵ null ;
              add t_i to the end of sequence ;
              minSlack ⟵ ST(t_i);
          else if TF(t_i) = minSlack then
              if sequence.lastItem is imm_pred of t_i then
                  add t_i to the end of sequence
              end
          end
      end
      /* distribute deadline over some partitions                      */
      CST ⟵ max_{t∈imm_Preds(Sequence.firstItem)} LFT(t);
      CFT ⟵ LFT(sequence.lastItem) ;
      Distribute deadline (CFT − CST) over all partitions in sequence
       proportional to MET(t_i) ;
  end
```

(sub-deadline) of a task, it is sufficient to only add time interval assigned to the task to its SST. SST of the next task in the chain is also equal to SFT of the previous task in that chain. It turns out that SFT of the last task in the chain is equal to CFT. We mark all the tasks in the chain as assigned tasks and update unassigned tasks' EST and LFT measures and then continue with selecting a new *critical chain*.

This procedure goes on until all tasks are assigned. It should be mentioned that for each assigned task, EST and LFT are considered as SST and SFT, respectively. By assigning time window to the last task, sub-deadline assignment procedure is completed.

Finding computational complexity of the proposed deadline distribution algorithm is beyond the scope of this paper, but intuitively, it can be seen that in the worst-case, the algorithm assigns sub deadline to at least one task in each iteration. Moreover, the complexity of each iteration for updating SST and SFT is $O(n^2)$, where n is the number of tasks in the workflow. Therefore, the computational complexity of the algorithm is $O(n^3)$, but the upper bound can be more tight than stated. This bound is pessimistic and the practical complexity for real applications is much less than this value. From another point of view, we can consider that the complexity of updating SST, SFT and ST is $O(e)$, where e is the number of dependency relations between the tasks, or the number of edges in DAG representing of the workflow. Therefore, the computational complexity of the algorithm is $O(ne)$. Since e can take $(n)(n+1)/2$ in worst case, the computational complexity of our proposed algorithm is $O(n^3)$.

4.3 An Illustrative Example

Figure 2 is an example of *critical chain* deadline distribution for workflow application shown in Figure 1 while a sample grid environment with the specification presented in Table 1 is considered. Each processing node in the sample grid has its own specific processing power indicating million instructions that the node can process per second (MIPS). Obviously, different processing nodes have different usage prices based on their processing speeds. In this sample, bandwidths of all links among the servers are assumed to be 512 Mbps. In Figure 2 A), MET and MTT measures are shown for each task. As an example, task t_2 has 426 000 MI with computation requirement type of B. The fastest server which can execute this task is server s_2. Therefore, MET(t_2) is 426 000 MI/2 000 MIPS = 213 S. Similarly, we can compute MTT and MET measures for all tasks.

We can compute sub-deadlines for this example in three iterations shown in Figure 2 B) to Figure 2 D). The measures EST, LST and ST are computed for all tasks, and *critical chain* is obtained (e.g., $\{t_0, t_2, t_4, t_5\}$). This step is shown in Figure 2 B). In the first iteration, chain interval is equal to whole deadline (1250). Since, task t_2 needs 225 seconds to be completed (MET + MTT = 213 s + 12 s = 225 s), and task t_4 needs 125 seconds, the time assigned to task t_2 is 225/125 = 9/5 times greater than the time assigned to task t_4. Similarly, the time interval will be distributed over all members of the *critical chain*. Iteratively, we update EST, LST and ST measures to obtain another *critical chain* (chain $\{t_1\}$ in Figure 2 C)). Since, there exists only one task in *critical chain* shown in Figure 2 C), the whole time interval is assigned to task t_1, and SST and SFT measures of task t_1 are set to 50 and 500, respectively (Figure 2 D)). Finally, SST and SFT measures related to task t_3 are forced to be 500 and 750, respectively.

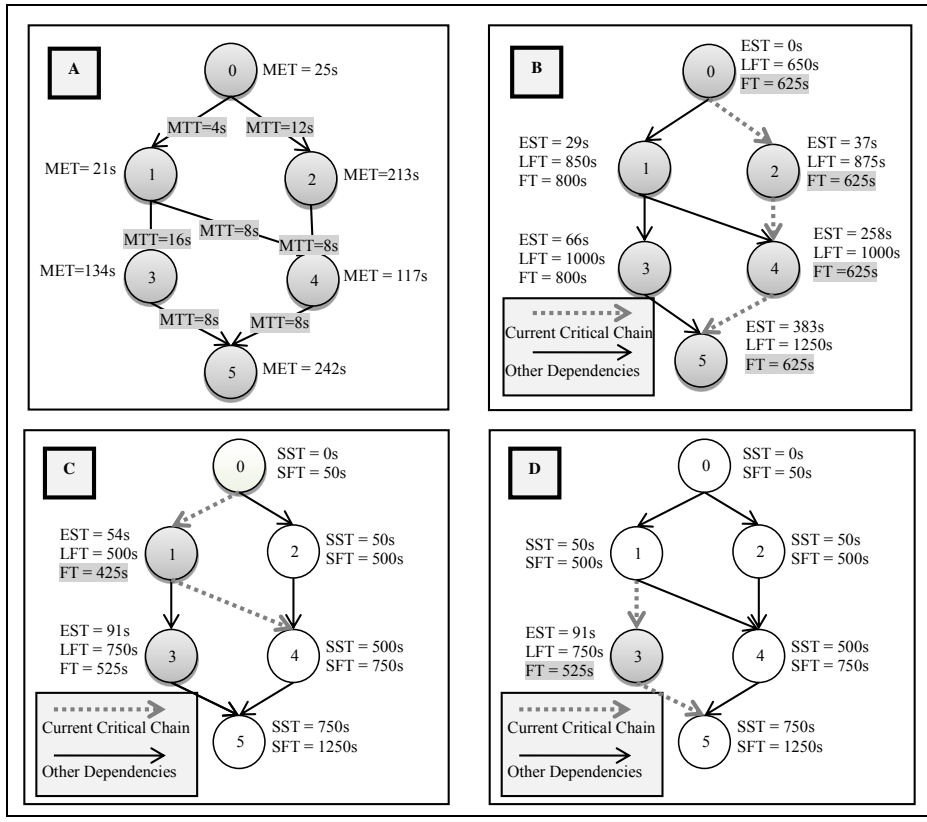

Figure 2. Deadline distribution over a sample workflow

4.4 Partitioning Technique

Partitioning is a method to optimally solve branches with several sequential tasks in grid workflows [9]. In partitioning model, tasks are divided into two different categories: simple tasks and synchronization tasks. If a task has at most one immediate predecessor, and at most one immediate successor, then the task is named simple task, otherwise, it is called synchronization task. We use the word branch

Server	**Server Type**	**Processing Power (MIPS)**	**Price per Second** ($)
s_1	A, B, D	1 000	0.001
s_2	B, C, E	2 000	0.004
s_3	A, C, D, E	4 000	0.016

Table 1. Sample grid specification with three processing nodes

to refer to several sequential simple tasks (a sub-graph like a pipeline workflow). The mapping problem of branches can be solved using dynamic programming and hidden Markov model. In Markov model, each of the states can be considered as a triple like [*termination time, number of scheduled tasks in a branch, last scheduled server*]. Afterwards, one can calculate value of each state with the help of previous states using dynamic programming. In this problem, the value of each Markov state is the cost of scheduling. The detailed description of partitioning technique can be found in [9]. Moreover, it should be mentioned that partitioning does not take part in deadline distribution phase of the proposed algorithm. This technique is used besides mapping phase. First, we partition the workflow in the branches, and then, in the mapping phase, we allocate a resource to all tasks in the branch with dynamic programming if we encounter a branch.

It remains a subtle point that dynamic programming is very time consuming, and calculation time extremely increases with increasing the problem size. Size of this problem depends on three different factors: the number of branch tasks, number of servers, and termination time range. The number of branch tasks and servers are constant values and if those numbers get higher, dynamic programming becomes inefficient. In almost all cases in our problem, we encounter few task branches with limited number of servers which can process each task. Therefore, dynamic programming can appropriately tolerate those factors in our case. As mentioned earlier, the third factor is termination time. If the time interval is wide, then we can segment the time to larger time pieces. Actually, we can handle larger problems in exchange for a bit of accuracy using segmentation.

4.5 Mapping Based on Priority

In the mapping phase, we try to map servers to the tasks to optimize the overall cost besides meeting the overall deadline. To do this, it is tried to have a local optimization in server allocation process to hopefully reach the global optimization. In the mapping phase, we iteratively pass over three steps. Mapping algorithm firstly selects a ready task (partition), a task that all of its immediate predecessors have been executed. Secondly, it updates the start time of unscheduled tasks to the minimum finish time of immediate predecessor plus delay of incoming link of predecessor. When there are one or more processors for processing a task before expiring its deadline, we choose the cheapest processor among those. If there is no server available to fulfill task's deadline, a server with minimum deadline violation is selected. Since both the number and power of the servers existing in the system are limited, it is important to have a priority algorithm to select a ready task to be assigned to the limited servers. The following three different methods for priority-based mapping are used in this paper:

Simple priority: A ready task with lower task ID has higher priority for scheduling in this method. Using this mechanism, we can simulate random priority.

Start time priority: A ready task with minimum start time has higher priority during the mapping phase. The main idea of start time priority method is that if a task has a lower start time, then there are probably more tasks waiting for that task, so it should be scheduled as soon as possible. The other idea behind start time priority is that, as soon as there is a ready task available for scheduling, it may be better to schedule it and do not wait for other tasks to become ready.

Slack time priority: For each ready task, a measure named *slack time* is computed using SST and SFT measures. Equation (8) computes slack time for each task.

$$\mathrm{ST}(t_i) = \mathrm{SFT}(t_i) - \mathrm{SST}(t_i) - \mathrm{MET}(t_i). \tag{8}$$

In slack time priority, the ready task with minimum slack time is prior to be scheduled. The main idea of slack time priority method is that if a task has lower slack time, then scheduling this task is more likely to be critical, and postponing its scheduling may lead to local sub-deadline violation and, consequently, possible global deadline violation.

According to the results presented in Section 5, slack time priority shows better performance in comparison with two other approaches. Finally, for the sake of brevity, we only present the performance analysis of the slack time priority.

5 PERFORMANCE EVALUATION

Accurate performance evaluation not only depends on the perfect implementation of the proposed and benchmark algorithms, but also it highly relies on the test data and experimental setup. In this section, it is described how workflow test sets are generated, and what are the main experimental setups. After that, the results obtained from the experiments are presented.

5.1 Generating Workflows

Three types of common workflow structures, pipeline, parallel and hybrid workflows [9, 10, 11, 13, 14, 19, 37], are considered in this paper. These structures are shown in Figure 3. Pipeline workflow consists of numbers of tasks in a sequential order (Figure 3 A)). Parallel workflows include multiple pipelines with some middle synchronizer tasks (Figure 3 B)). Hybrid workflows are combinations of pipeline and parallel workflows in an arbitrary order (Figure 3 C)). Structure of pipeline workflows is simple, but many factors influence on construction of parallel and hybrid workflows. Most important factors in parallel workflow construction are the maximum width of graph (the number of parallel pipeline chains), and maximum number of possible sub-pipeline tasks. We set the maximum width and the maximum pipeline length to 10 and 20, respectively. There is no constraint on choosing other numbers for width and maximum pipeline length. We just choose those numbers according

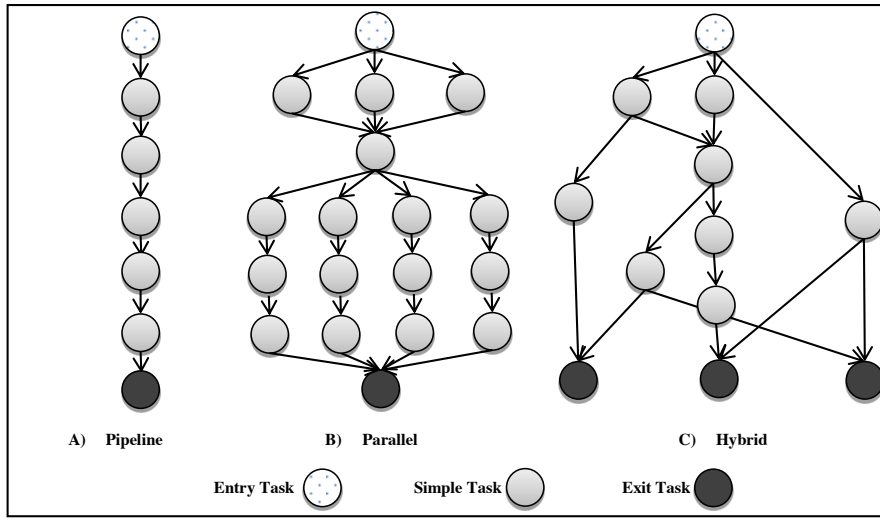

Figure 3. Different types of random workflow structures

to previously done similar work [10]. Generating hybrid workflow structures is more complex. There are some papers addressing this problem [37, 38, 39]. They study parametric features of each generation method such as average critical path length and average edge. In [39], a tool for random graph generation, named TGFF, which is approximately fair in graph generation was presented. According to study done by Cordeiro et al. [37], the Max-in/Max-out degree method is one of the best methods for workflow generation. Hence, TGFF is used in this paper to produce hybrid workflows with Max-in/Max-out method. For TGFF method, the Max-in (Max-out) parameters are considered from 1 to 3 ($Max_{in}(Max_{out}) \in \{1, 2, 3\}$), and the number of tasks varies from 30 to 1000. Figure 3 C) shows a sample output of TGFF with specified parameters. In addition to random workflows, we compare the proposed method with other algorithms applying the standard workflows used in [13] called CyberShake, Montage, LIGO, Gnome, and SIPHT, which are real workflows in the scientific or business communities.

5.2 Experimental Setup

We consider details of real environments in our experiments such as grid structure, network bandwidth, processing power of each processing node, largeness of workflows, deadline determination and so forth. All assumptions made in this work are consistent with previously done research work [6, 9, 10, 13, 14]. In this section, we have a glance on implementation details.

Workflow Specification: The structures of workflows are chosen randomly as illustrated in Section 5.1. From the viewpoint of granularity, we consider three

types of workflows: small, medium and large. The number of tasks vary from 30 to 200, from 200 to 600, and from 600 to 1 000 in small, medium and large workflows, respectively. For the heterogeneity of workflows, it is assumed that each task has specific processing requirements. We assume there are 15 different types of tasks. Moreover, the size of each task varies from 100 000 MI to 900 000 MI in this study. For each workflow, there are several exit tasks and only one entry task. Each task can be executed as soon as its corresponding resource receives the required data for processing that task. The input/output data for the task changes from 10 MB to 1 GB, as well.

Grid Specification: To simulate the server heterogeneity in grid network, we consider 15 different types of services, each service supported by 10 different grid servers. Network bandwidth between each two grid servers is considered to be in the range of 200 Mbps to 512 Mbps. Grid server processing power shown by MIPS (Million Instructions Per Second) varies from 1 000 MIPS to 5 000 MIPS for each node. For each processing node there is a price proportional to its power ranging from 0.001 \$ to 0.025 \$. This means that executing a task on a server with a processor n times faster than another, imposes n times more cost to the scheduling.

Hardware Specification: We run the simulation on a regular PC with Intel® Core™ i5-4200M (3M Cache, up to 3.10 GHz) and 4 GB RAM. Based on this hardware configuration, each test case runs in 1 to 2 seconds (except for huge pipeline workflows), and consequently, each test set containing more than 500 different test cases runs in less than 30 minutes. For a pipeline workflow test case with 200 tasks, it takes up to 5 minutes to run. It turns out that the time required to run the simulation reduces by using more powerful hardware.

Deadline Assignment: Each workflow is tested with 9 different deadlines, i.e., $\delta_n = T_{min} \times \theta$, where T_{min} is the minimum completion time for that workflow on the specified grid, and $\theta \in \{1, 1.05, 1.1, 1.15, 1.2, 1.5, 2, 2.5, 3\}$. Practically, there is no need to consider $\theta > 3$, because in that range, all tasks are delivered to lower and cheaper processing nodes, and approximately all methods show the same efficiency. Determining T_{min} is an NP-hard problem, so we use HEFT greedy algorithm [28] to estimate minimum completion time.

5.3 Result Analysis

We compare our method with four recent most-cited methods: PCP Fair, PCP Decrease, DTL, and DBL [9, 13, 29]. As mentioned earlier, we use three types of workflows to do experiments. Each of random workflows is tested by over 800 instances to achieve more reliable results. Each instance is tested by 5 different methods with 8 different deadlines, and the experiment is done with near 100 000 iterations. Three different priority-based mapping methods are tested, but for the sake of brevity, only one of the results is reported here.

		DBL	DTL Decrease	PCP Fair	PCP	Critical Chain
PipeLine	**Small**	0	0	0	0	100
	Medium	0	0	0	0	100
	Large	0	0	0	0	100
Parallel	**Small**	10.4	10.5	0.33	0.22	89
	Medium	11.88	11.78	2.78	1.56	83.69
	Large	17	17	0.22	13.44	69.33
Hybrid	**Small**	26.02	25.4	0.34	11.23	48.87
	Medium	20.09	22.10	2.91	22.27	31.58
	Large	29.89	24.66	0	16.31	36.62

Table 2. Best result percentage

In our study, an algorithm is superior to another if it appropriately satisfies the following three properties:

Best results: if results of two different schedulings violate the deadline, the result that has the minimum time is the best result for our case study. If none of them has deadline violation, the result with the minimum scheduling price is the best one. An algorithm with maximum percentage of best results is named best suitable algorithm in our study.

Deadline Violation Rate (DVR): if an algorithm has minimum deadline violation rate, then it is superior to the others.

Average cost: the algorithm with minimum average cost is preferred to the others.

The best result percentages of all algorithms for all test sets are given in Table 2. As it can be seen in Table 2, from the aspect of the best results percentage, the proposed *critical chain* method is dominant. This means that for most of the applications, *critical chain* shows the best performance compared to the other benchmarks.

In pipeline workflows, *critical chain* uses dynamic programming for branches, and computes optimal solution; therefore, this method is dominant in this type of workflows. Since the algorithm produces best result in pipeline workflows, the percentage of the test cases on which any other method can produce best result is *zero*. This is shown in Table 2 by elements 0 inside cells related to the pipeline workflows. It should be mentioned that since the scheduling problem considered in this paper belongs to the set of NP-hard problems, and there is no polynomial solution to solve it, we tried to solve it by dynamic programming that is very time consuming. Therefore, the best solution with 100% confidence in pipeline workflows is achieved by imposing extra scheduling time on the scheduler. Dynamic programming imposes heavy computational overhead to the scheduler, so that, computing the best solution for over 200 sequential tasks in a reasonable time is impractical for schedulers. For the parallel workflows, our algorithm is dominant too with a slight decrease in performance in comparison with other algorithms. Parallel workloads

contain many pipeline chains and our algorithm uses dynamic programming and fair deadline distribution to receive this performance. If we consider a very long length for a parallel workload, then we have to solve many long pipeline problems to schedule the long-length parallel workload, and as a result, we may encounter the performance problem described above. It is worthwhile to mention that scheduling parallel workloads has no polynomial optimal solution. Therefore, performance degradation in comparison with pipeline workflows is reasonable. In the hybrid workflows, the pipeline chains are in minority, so dynamic programming cannot be useful further, but our algorithm still shows relatively good performance. We believe that our deadline distribution method has an important role to achieve this. Based on the simulations done, we found that increasing the size of the graph does not considerably change the results. Hence, the proposed algorithm shows the same performance for different workflow sizes. In other words, the structure of a workflow has much greater impact on the results. We continue to analyze the algorithms in the view of deadline violation rate and average cost properties for each of the workflow structures. For the sake of brevity, we depict only two classes of diagrams for each workflow type. The left-side diagrams in Figure 4 show average cost resulted by algorithms. Since these diagrams show the average cost for all small, medium, and large applications, they should be normalized to match with each other in a single diagram. The Normalized Cost (NC) is computed as Equation (9).

$$\mathrm{NC} = \frac{\text{Scheduling Cost}}{\text{Minimum Scheduling Cost}}. \tag{9}$$

After scheduling each workflow, the cost resulting from the scheduling is divided by the minimum scheduling cost obtained from a greedy algorithm and then NC is achieved. The minimum scheduling cost is obtained from greedy scheduler that assigns each task to the cheapest resource with unlimited deadline. The right-side diagrams in Figure 4 show the deadline violation rate of the algorithms. If a violation rate of one algorithm is α with β as a deadline factor, it means that α is the probability of deadline violation by applying that scheduling algorithm with $\beta \times T_{min}$ as a deadline. All algorithms try to schedule all tasks to faster servers whenever deadlines are firm (deadline = 1). This tends to increase the scheduling cost resulted from all methods. Moreover, the rate of deadline violation increases in this situation. Expanding the deadline associated with each task, and as a result, increasing the overall deadline of the workflow, deadline violation rate and execution cost resulted from all algorithms decrease.

For the pipeline (Figure 4 A) and Figure 4 B)) and parallel (Figure 4 C) and Figure 4 D)) workflows, DVR and NC for *critical chain* get minimum values compared to the other algorithms. This shows that the best method for scheduling these types of workflows amongst all of the algorithms implemented in this paper is *critical chain*. For hybrid workflows (Figure 4 E) and Figure 4 F)) DVR in all deadlines and NC for soft deadlines are minima in *critical chain* method, but in the firm deadlines, NC for *critical chain* is slightly more than DBL algorithm. It is

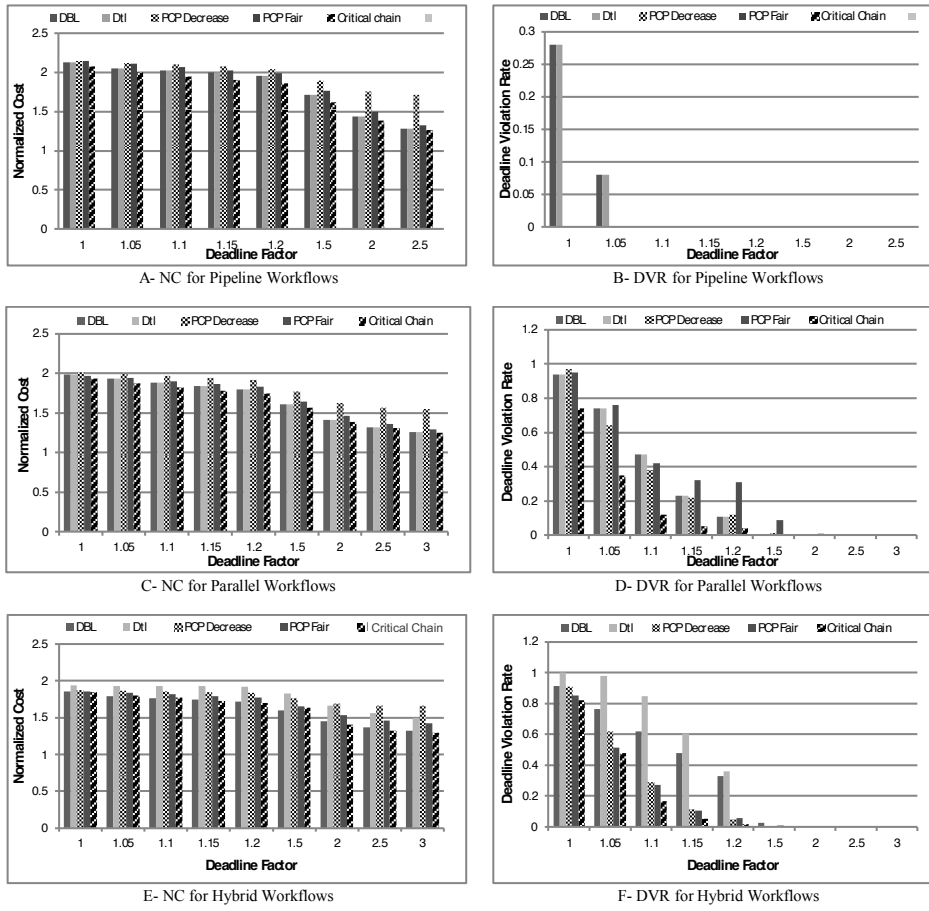

Figure 4. Comparison of Normalized Cost (NC) and Deadline Violation Rate (DVR) resulted from all algorithms for random workflows

predictable, because in a firm deadline, *critical chain* tries to not exceed deadline by scheduling tasks on the faster (consequently more expensive) resources. Finally, it can be stated that DVR and average NC for *critical chain* are minima in comparison with other algorithms. Hence, the proposed *critical chain* method outperforms previous methods from the viewpoint of best results, deadline violation rate, and average cost.

As mentioned in Section 5.1, we apply our proposed algorithm on realistic standard workflows presented in [13]. The results obtained from applying the proposed algorithm on medium size workflows including CyberShake, Montage, and LIGO are presented in Figure 5, and similarly the results gained from workflows with large and small sizes, Gnome and SIPHT, are presented in Figure 6. As can be seen in

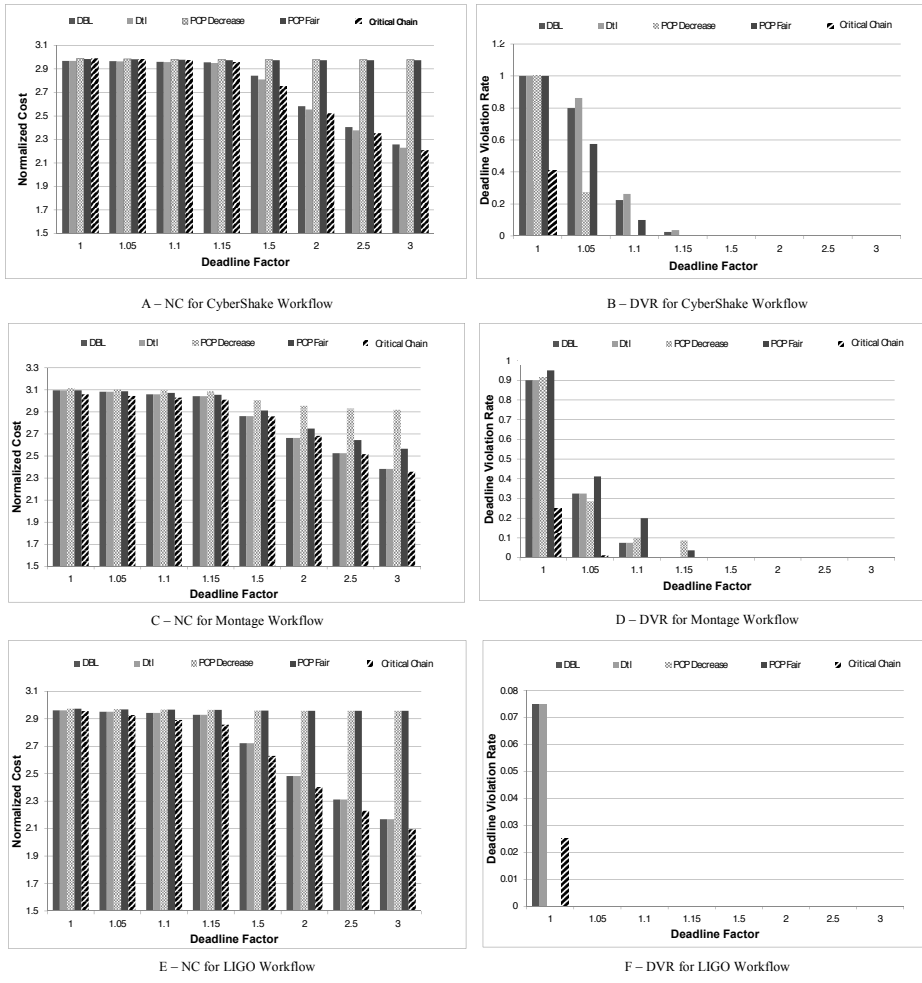

Figure 5. Comparison of Normalized Cost (NC) and Deadline Violation Rate (DVR) resulted from all algorithms for standard benchmark workflows with medium sizes

both Figure 5 and Figure 6, our algorithm dominates other algorithms in average by minimizing DVR and reducing the scheduling cost.

6 CONCLUSIONS AND FUTURE WORK

The basic principle used in utility computing and grid computing is identical which is providing computational resources as a service. One of the most important problems in such environments is scheduling deadline constrained bag-of-tasks applications on computational resources. In this paper, we study the deadline of applications as one

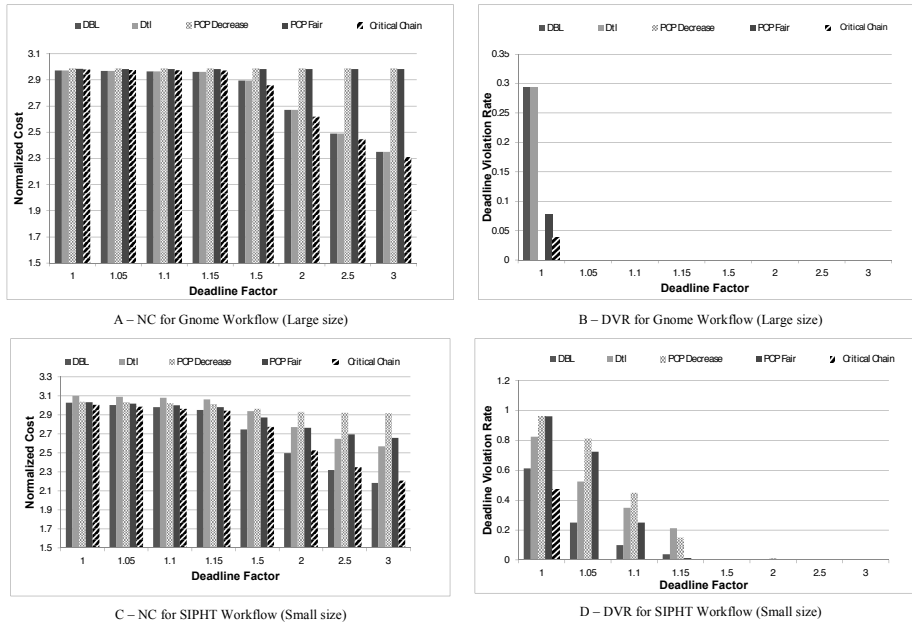

Figure 6. Comparison of Normalized Cost (NC) and Deadline Violation Rate (DVR) resulted from all algorithms for standard benchmark workflows with large and small sizes

of the more interesting factors in grid computing, and try to deliver a service with specified QoS and minimum cost. Therefore, our problem can be considered as a time-cost trade-off problem. To solve this optimization problem, two-step *critical chain* heuristic is presented. In deadline distribution phase, the algorithm applies a fairer mechanism to better deadline distribution, which finally, leads to lower cost of service. In the second step named resource allocation phase, resources are allocated to the tasks efficiently according to the priority of tasks. Finally, applying the proposed method to different scenarios and system settings, it is shown that the proposed approach outperforms other similar existing methods.

Critical chain is applicable in other distributed computing domains such as clouds. In cloud systems, customers can select their desired service based on their budget and time constrains, and pay for using these services. Therefore, the problem of scheduling workflow applications on limited resources considering time and budget constraints is an interesting problem in clouds. So, one possible extension to this work is applying the proposed critical chain method on scheduling workflows in cloud infrastructures considering their specific characteristics and requirements.

Critical path-based deadline distribution has deficiency in bursting tasks. In other words, if the workflow width becomes unpredictable during the specified time period, the deadline distribution based on critical path with any available method would be inefficient. So, one can modify the proposed method to handle this type

of workflows. Furthermore, considering the map-reduce structure as a new type of workflows and modifying the proposed method to have high performance in map-reduce workflows can be assumed as another objective. Evaluating other performance measures such as resource utilization and taking server setup cost into account can be considered as an important direction for improvements of the current work.

REFERENCES

[1] Foster, I.—Kesselman, C.: The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[2] Czajkowski, K.—Fitzgerald, S.—Foster, I.—Kesselman, C.: Grid Information Services for Distributed Resource Sharing. Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, San Francisco, CA, USA, August 2001, pp. 181–194, doi: 10.1109/HPDC.2001.945188.

[3] Broberg, J.—Venugopal, S.—Buyya, R.: Market-Oriented Grids and Utility Computing: The State-of-the-Art and Future Directions. Journal of Grid Computing, Vol. 6, 2008, No. 3, pp. 255–276, doi: 10.1007/s10723-007-9095-3.

[4] Toporkov, V.—Yemelyanov, D.—Potekhin, P.—Toporkova, A.—Tselishchev, A.: Metascheduling and Heuristic Co-Allocation Strategies in Distributed Computing. Computing and Informatics, Vol. 34, 2015, No. 1, pp. 45–76.

[5] Yu, J.—Buyya, R.—Ramamohanarao, K.: Workflow Scheduling Algorithms for Grid Computing. In: Xhafa, F., Abraham, A. (Eds.): Metaheuristics for Scheduling in Distributed Computing Environments. Springer, Berlin, Heidelberg, Studies in Computational Intelligence, Vol. 146, 2008, pp. 173–214.

[6] Yu, J.—Buyya, R.: A Budget Constrained Scheduling of Workflow Applications on Utility Grids Using Genetic Algorithms. The Workshop on Workflows in Support of Large-Scale Science, Paris, France, June 2006, pp. 1–10, doi: 10.1109/WORKS.2006.5282330.

[7] Dong, F.—Akl, S. G.: Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. Technical Report No. 2006-504: School of Computing, Queen's University, Kingston, Ontario, 2006, pp. 1–55.

[8] Atanak, M. M.—Dogan, A.: Improving Real-Time Data Dissemination Performance by Multi Path Data Scheduling in Data Grids. Computing and Informatics, Vol. 34, 2015, No. 2, pp. 402–424.

[9] Yu, J.—Buyya, R.—Tham, C. K.: Cost-Based Scheduling of Scientific Workflow Applications on Utility Grids. Proceedings of the 1st International Conference on e-Science and Grid Computing, Melbourne, Australia, December 2005, pp. 140–147.

[10] Yuan, Y.—Li, X.—Wang, Q.—Zhu, X.: Deadline Division-Based Heuristic for Cost Optimization in Workflow Scheduling. Information Sciences, Vol. 179, 2009, No. 15, pp. 2562–2575.

[11] Rahman, M.—Venugopal, S.—Buyya, R.: A Dynamic Critical Path Algorithm for Scheduling Scientific Workflow Applications on Global Grids. Proceed-

ings of the 3rd IEEE International Conference on e-Science and Grid Computing (e-Science 2007), Bangalore, India, December 2007, pp. 35–42, doi: 10.1109/E-SCIENCE.2007.3.

[12] Alkhanak, E. N.—Lee, S. P.—Khan, S. U. R.: Cost-Aware Challenges for Workflow Scheduling Approaches in Cloud Computing Environments: Taxonomy and Opportunities. Future Generation Computer Systems, Vol. 50, 2015, pp. 3–21, doi: 10.1016/j.future.2015.01.007.

[13] Abrishami, S.—Naghibzadeh, M.—Epema, D. H. J.: Cost-Driven Scheduling of Grid Workflows Using Partial Critical Paths. Proceedings of the 11th IEEE/ACM International Conference on Grid Computing, Brussels, Belgium, October 2010, pp. 81–88, doi: 10.1109/GRID.2010.5697955.

[14] Yao, Y.—Liu, J.—Ma, L.: Efficient Cost Optimization for Workflow Scheduling on Grids. Proceedings of the International Conference on Management and Service Science, Wuhan, China, August 2010, pp. 1–4, doi: 10.1109/ICMSS.2010.5577645.

[15] Henzinger, T. A.—Singh, A. V.—Singh, V.—Wies, T.—Zufferey, D.: FlexPRICE: Flexible Provisioning of Resources in a Cloud Environment. Proceedings of the 3rd IEEE International Conference on Cloud Computing, Miami, USA, July 2010, pp. 83–90, doi: 10.1109/CLOUD.2010.71.

[16] Arabnejad, H.—Barbosa, J. G.—Prodan, R.: Low-Time Complexity Budget–Deadline Constrained Workflow Scheduling on Heterogeneous Resources. Future Generation Computer Systems, Vol. 55, 2016, pp. 29–40, doi: 10.1016/j.future.2015.07.021.

[17] Rahman, M.—Hassan, R.—Ranjan, R.—Buyya, R.: Adaptive Workflow Scheduling for Dynamic Grid and Cloud Computing Environment. Concurrency and Computation: Practice and Experience, Vol. 25, 2013, No. 13, pp. 1816–1842.

[18] Kwok, Y. K.—Ahmad, I.: Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, Vol. 7, 1996, No. 5, pp. 506–521.

[19] Lin, M.—Lin, Z.: A Cost-Effective Critical Path Approach for Service Priority Selections in Grid Computing Economy. Decision Support Systems, Vol. 42, 2006, No. 3, pp. 1628–1640.

[20] Tabbaa, N.—Entezari-Maleki, R.—Movaghar, A.: Reduced Communications Fault Tolerant Task Scheduling Algorithm for Multiprocessor Systems. Procedia Engineering, Vol. 29, 2012, pp. 3820–3825, doi: 10.1016/j.proeng.2012.01.577.

[21] Yang, T.—Gerasoulis, A.: DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Transactions on Parallel and Distributed Systems, Vol. 5, 1994, No. 9, pp. 951–967.

[22] Foster, I.—Fidler, M.—Roy, A. J.—Sander, V.—Winkler, L.: End-to-End Quality of Service for High-End Applications. Computer Communications, Vol. 27, 2004, No. 14, pp. 1375–1388, doi: 10.1016/j.comcom.2004.02.014.

[23] Dogan, A.—Ozguner, F.: Scheduling Independent Tasks with QoS Requirements in Grid Computing with Time-Varying Resource Prices. In: Parashar, M. (Ed.): Grid Computing – GRID 2002. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2536, 2002, pp. 58–69.

[24] Tabbaa, N.—Entezari-Maleki, R.—Movaghar, A.: A Fault Tolerant Scheduling Algorithm for DAG Applications in Cluster Environments. In: Snasel, V., Platos, J., El-Qawasmeh, E. (Eds.): Digital Information Processing and Communications (ICDIPC 2011). Springer, Berlin, Heidelberg, Communications in Computer and Information Science, Vol. 188, 2011, pp. 189–199.

[25] Entezari-Maleki, R.—Movaghar, A.: A Genetic-Based Scheduling Algorithm to Minimize the Makespan of the Grid Applications. In: Kim, T. H., Yau, S. S., Gervasi, O., Kang, B. H., Stoica, A., Ślęzak, D. (Eds.): Grid and Distributed Computing, Control and Automation (GDC 2010, CA 2010). Springer, Berlin, Heidelberg, Communications in Computer and Information Science, Vol. 121, 2010, pp. 22–31.

[26] Kardani-Moghaddam, S.—Khodadadi, F.—Entezari-Maleki, R.—Movaghar, A.: A Hybrid Genetic Algorithm and Variable Neighborhood Search for Task Scheduling Problem in Grid Environment. Procedia Engineering, Vol. 29, 2012, pp. 3808–3814, doi: 10.1016/j.proeng.2012.01.575.

[27] Agrawal, K.—Benoit, A.—Magnan, L.—Robert, Y.: Scheduling Algorithms for Linear Workflow Optimization. Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Atlanta, GA, April 2010, pp. 1–12, doi: 10.1109/IPDPS.2010.5470346.

[28] Zhao, H.—Sakellariou, R.: An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In: Kosch, H., Böszörményi, L., Hellwagner, H. (Eds.): Euro-Par 2003 Parallel Processing. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2790, 2003, pp. 189–194.

[29] Yuan, Y.-C.—Li, X.—Wang, Q.—Zhang, Y.: Bottom Level Based Heuristic for Workflow Scheduling in Grids. Chinese Journal of Computers, Vol. 31, 2008, No. 2, pp. 280–290.

[30] Poola, D.—Garg, S. K.—Buyya, R.—Yang, Y.—Ramamohanarao, K.: Robust Scheduling of Scientific Workflows with Deadline and Budget Constraints in Clouds. Proceedings of the IEEE 28th International Conference on Advanced Information Networking and Applications, Victoria, Canada, May 2014, pp. 858–865, doi: 10.1109/AINA.2014.105.

[31] Calheiros, R. N.—Buyya, R.: Cost-Effective Provisioning and Scheduling of Deadline-Constrained Applications in Hybrid Clouds. In: Wang, X. S., Cruz, I., Delis, A., Huang, G. (Eds.): Web Information Systems Engineering (WISE 2012). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7651, 2012, pp. 171–184.

[32] Qiu, X.—Yeow, W. L.—Wu, C.—Lau, F. C. M.: Cost-Minimizing Preemptive Scheduling of Mapreduce Workloads on Hybrid Clouds. Proceedings of the IEEE/ACM 21st International Symposium on Quality of Service (IWQoS), Montreal, Canada, June 2013, pp. 1–6.

[33] Van den Bossche, R.—Vanmechelen, K.—Broeckhove, J.: Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads. Proceedings of the IEEE 3rd International Conference on Cloud Computing, Miami, FL, USA, July 2010, pp. 228–235, doi: 10.1109/CLOUD.2010.58.

[34] Wang, M.—Zhu, L.—Zhang, Z.: Risk-Aware Intermediate Dataset Backup Strategy in Cloud-Based Data Intensive Workflows. Future Generation Computer Systems, Vol. 55, 2016, pp. 524–533, doi: 10.1016/j.future.2014.08.009.

[35] Fard, H. M.—Prodan, R.—Fahringer, T.: A Truthful Dynamic Workflow Scheduling Mechanism for Commercial Multicloud Environments. IEEE Transactions on Parallel and Distributed Systems, Vol. 24, 2013, No. 6, pp. 1203–1212.

[36] De, P.—Dunne, E. J.—Ghosh, J. B.—Wells, C. E.: The Discrete Time-Cost Tradeoff Problem Revisited. European Journal of Operational Research, Vol. 81, 1995, No. 2, pp. 225–238.

[37] Cordeiro, D.—Mounié, G.—Perarnau, S.—Trystram, D.—Vincent, J.-M.—Wagner, F.: Random Graph Generation for Scheduling Simulations. Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools '10), Torremolinos, Spain, March 2010, pp. 60:1–60:10, doi: 10.4108/ICST.SIMUTOOLS2010.8667.

[38] Li, H.: Workload Characterization, Modeling, and Prediction in Grid Computing. Doctoral Thesis. LIACS, Computer Systems Group, Faculty of Science, Leiden University, 2008.

[39] Dick, R. P.—Rhodes, D. L.—Wolf, W.: TGFF: Task Graphs for Free. Proceedings of the 6th International Workshop on Hardware/Software Codesign (CODES/CASHE '98), Seattle, USA, March 1998, pp. 97–101.

Alireza Dehlaghi-Ghadim is currently Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Tehran, Tehran, Iran. He received his M.Sc. degree from Sharif University of Technology, Tehran, Iran, in 2012, and his B.Sc. degree from Iran University of Science and Technology, Tehran, Iran in 2009. His main research interests are grid and cloud computing, task scheduling algorithms, load balancing and resource allocation methods.

Reza Entezari-Maleki is Post-Doctoral Researcher in the School of Computer Science at Institute for Research in Fundamental Sciences (IPM) in Tehran, Iran. He received his Ph.D. in computer engineering (software discipline) from Sharif University of Technology, Tehran, Iran in 2014, and M.Sc. and B.Sc. degrees in computer engineering (software discipline) from Iran University of Science and Technology, Tehran, Iran in 2009 and 2007, respectively. He visited the Seoul National University in Seoul, South Korea, Duke University in NC, USA, and Instituto Superior Técnico in Lisbon, Portugal in 2012, 2013, and 2015, respectively, His main research interests are performance/dependability modeling and evaluation, grid and cloud computing, and task scheduling algorithms.

Ali Movaghar is Professor in the Department of Computer Engineering at Sharif University of Technology in Tehran, Iran and has been on the Sharif faculty since 1993. He received his B.Sc. degree in electrical engineering from the University of Tehran in 1977, and M.Sc. and Ph.D. degrees in computer, information, and control engineering from the University of Michigan, Ann Arbor, in 1979 and 1985, respectively. He visited the Institut National de Recherche en Informatique et en Automatique in Paris, France and the Department of Electrical Engineering and Computer Science at the University of California, Irvine in 1984 and 2011, respectively, worked at AT & T Information Systems in Naperville, IL in 1985–1986, and he was teaching at the University of Michigan, Ann Arbor in 1987–1989. His research interests include performance/dependability modeling and formal verification of wireless networks and distributed real-time systems. He is a senior member of the IEEE and the ACM.