Deadline Constrained Scientific Workflow Scheduling on Dynamically Provisioned Cloud Resources

Vahid Arabnejad, Kris Bubendorfer, Bryan Ng

*School of Engineering and Computer Science, Victoria University of Wellington, New Zealand*

**Abstract**

Commercial cloud computing resources are rapidly becoming the target platform on which to perform scientific computation, due to the massive leverage possible and elastic pay-as-you-go pricing model. The cloud allows researchers and institutions to only provision compute when required, and to scale seamlessly as needed. The cloud computing paradigm therefore presents a low capital, low barrier to operating dedicated HPC eScience infrastructure. However, there are still significant technical hurdles associated with obtaining sufficient execution performance while limiting the financial cost, in particular, a naive scheduling algorithm may increase the cost of computation to the point that using cloud resources is no longer a viable option.

The work in this article concentrates on the problem of scheduling deadline constrained scientific workloads on dynamically provisioned cloud resources, while reducing the cost of computation. Specifically we present two algorithms, Proportional Deadline Constrained (PDC) and Deadline Constrained Critical Path (DCCP) that address the workflow scheduling problem on such dynamically provisioned cloud resources. These algorithms are additionally extended to refine their operation in task prioritization and backfilling respectively. The results in this article indicate that both PDC and DCCP algorithms achieve higher cost efficiencies and success rates when compared to existing algorithms.

*Keywords:* Scheduling, Cloud Computing, Resource Allocation, Scientific Workflows, Resource Provisioning.

*Email addresses:* Vahid.Arabnejad@ecs.vuw.ac.nz (Vahid Arabnejad), kris@ecs.vuw.ac.nz (Kris Bubendorfer), bryan.ng@ecs.vuw.ac.nz (Bryan Ng)
1. Introduction

Elastic, on demand cloud computing enables significant computational leverage to be applied to real world problems, be they medical, commercial, industrial or scientific. From a scheduling point of view, the most interesting subset of these problems are those that can be represented as workflows. Workflows are being extensively used by scientists to model and to manage complex, compute and data intensive experiments [1], and more of these workflows are being progressively moved onto commercial clouds [2, 3, 4, 5, 6].

Typical commercial cloud services charge on the basis of the number of hours the resources (such as CPU, network bandwidth and amount of storage) are used. This charging model is referred to as pay-per-use. Other advantages of using commercial clouds for scientific computation include reliability and fault tolerance, and access to specialized resources such as GPUs. The flexibility inherent in the elastic cloud model, while powerful, may also result in inefficient usage and high costs when inadequate scheduling and provisioning decisions are made [7].

This article focuses on extended versions of two related algorithms from our recent work, Proportional Deadline Constrained (PDC) [8] and Deadline Constrained Critical Path (DCCP) [9]. Both algorithms belong to the class of list-based scheduling algorithms [10] consisting of a task prioritization phase and a task assignment phase. The PDC algorithm maximizes parallelism in a workflow by separating it into logical levels and then proportionally subdividing the overall workflow deadline over different levels. In this work we further improve the performance of the PDC algorithm by refining the task ranking carried out during the task prioritization step. The DCCP algorithm uses the concept of Constrained Critical Paths (CCP) to execute a set of tasks on the same instance with the goal of reducing communication cost between instances. In this work we extend the DCCP algorithm to strategically backfill left over capacity (residuals) in provisioned instances and apply several different backfilling strategies. This approach is most effective in data intensive workflows due to the reduction in data movement. In addition we also conduct a considerably more comprehensive and detailed evaluation of both algorithms to better understand the limitations on their performance.

We evaluate the algorithms using a CloudSim [11] simulation that features dynamically provisioned cloud resources and a pay-per-use model de-
rived from Amazon’s EC2 pricing model. The simulations were conducted using five scientific workflows generated by the pegasus workflow generator, Montage, SIPHT, LIGO, Cybershake and Epigenomics and each of these generated workflows consisted of 1000 tasks.

We compare the performance of the extended PDC and DCCP against two previously published algorithms (IC-PCP [12] and JIT [13]) by measuring success rate, normalized cost, and throughput. In terms of cost, PDC and DCCP algorithms generally returned the lowest compute costs over all workflows and instance configurations. Of particular note, for the Cybershake workflow, PDC returned costs approximately 10% of those incurred by JIT.

The remainder of this article is organized as follows: Section 2 gives an overview of existing approaches to scheduling workflows. In Section 3, we define the workflow scheduling problem and describe the basis of our algorithms with a graph model. Section 4 describes the two extended DCCP and the PDC algorithms. The cost and success rate of the different scheduling algorithms are evaluated and the results are extensively discussed in Section 5. Finally, we summarize our work in Section 6.

2. Related Work

Various algorithms based on heuristic, search-based, and meta-heuristic strategies, have been proposed for efficient resource scheduling – where tasks are either considered independent (bag of tasks) or dependent (workflows). As a well studied problem there are several comprehensive reviews of workflow scheduling methods in distributed environments [14, 15, 16]. Scheduling workflow tasks to resources while meeting workflow dependencies and constraints is known as the Workflow Scheduling Problem (WSP) – a class of problem that is known to be NP-complete [17]. When multiple Quality of Service (QoS) parameters are used as constraints, the problem of workflow scheduling becomes even more challenging. We will focus on heuristic approaches to solving the WSP. The reason is, that while search-based and meta-heuristic strategies produce acceptable answers, in a dynamic scheduling environment the need for a training, or initialisation phase, limits their use in practice.

Allocating workflow tasks to resources can be separated into two stages, the first being scheduling and the second is provisioning [14]. Given a set
of resources, the workflow task scheduling phase aims to determine the optimal execution order and task placement with respect to user and workflow constraints \[18, 19\]. The resource provisioning phase aims to determine the number and type of resources required and then to reserve these resources for workflow execution \[20, 21\].

While the majority of cloud scheduling systems necessarily include both scheduling and provisioning stages, prior research tends to focus on the scheduling phase, under the assumption that a pre-identified pool of (often homogenous) resources are used for execution, and with the goal of optimizing workflow execution time (makespan) without considering resource cost. In a commercial cloud environment this set of assumptions no longer holds.

In this section, we classify recent related research into three categories: Budget Constrained, Deadline Constrained and Deadline and Budget Constrained.

2.1. Budget Constrained Scheduling

A budget is the maximum amount of financial resource that users wish to pay to run their workflows. Algorithms in the budget constrained category attempt to minimize workflow completion time for a given budget.

In \[19\] Sakellariou et al. presented two budget constrained algorithms, LOSS and GAIN. The algorithms start with one of two different initial assignments. The first assignment is the Best assignment: a time optimized assignment in which the execution time is the minimum possible. For example, the HEFT algorithm \[21\] is used as an initial assignment for the LOSS algorithm. HEFT is one of the most common scheduling algorithms and attempts to reduce the workflow makespan. The second assignment is the Cheapest assignment: a cost-optimized assignment wherein all tasks are assigned to resources having the least execution cost. For example, GAIN uses the cheapest assignment as the initial assignment. Tasks are repeatedly selected for reassignment until the user constrained budget is reached. These algorithms however were designed for non elastic, grid environments.

An extension of the HEFT \[21\] algorithm called the Cost Conscious Scheduling Heuristic (CCSH) was presented by Li et al. in \[22\]. The CCSH first constructs a priority list of tasks and then assigns the task with the highest priority value to the most cost-efficient virtual machine (VM). However, only one VM type and one pricing model is considered. The authors later introduced the Pareto dominance cost efficient heuristic to the CCSH to consider different cost models \[23\].
In [24], Zeng et al. presented a budget-aware backtracking algorithm for executing large scale many task workflows, referred to as ScaleStar. Their algorithm uses a new metric termed the Comparative Advantage (CA) to select resources in a way to minimize cost. The CA metric attempts to balance cost and execution time. This work while developed for cloud, used a grid cost model.

2.2. Deadline Constrained Scheduling

The deadline is the time by which a workflow must complete its execution. In most clouds, heterogeneous resources are available that offer different levels of performance at different price offering. Generally, faster resources are more expensive in comparison with slower ones. Therefore, there is often an exploitable trade off between execution time and the cost of resources. The related work in this section is the closest in principles to our work on PDC and DCCP.

In [25], Yuan et al. presented the Deadline Early Tree (DET) algorithm. In DET, tasks are partitioned into two types: critical and non-critical activities. All tasks on the critical path are scheduled using dynamic programming under a given deadline. Non-critical tasks are backfilled between critical tasks. However, the communication time between tasks in a workflow is not taken into account by the DET scheduler.

The Hybrid Cloud Optimized Cost (HCOC) scheduling algorithm by Bittencourt and Madeira presented in [26] focuses on optimizing cloud-bursting from private to public clouds. The initial schedule starts to execute tasks on private cloud resources; if the initial schedule cannot meet the deadline, additional resources are leased from a public cloud on demand. The combination of private and public cloud models means that this work cannot be applied to a purely commercial cloud context.

In [12], Abrishami et al. presented the Infrastructure as a service (IaaS) Cloud Partial Critical Paths (IC-PCP). All tasks in a partial critical path (PCP) are scheduled to the same cheapest instance that can complete them by the given deadline. This avoids incurring communication costs for each PCP. However, the IC-PCP algorithm does not consider the boot and deployment time of VMs, even though these are created on demand. One extension of IC-PCP that attempts to further reduce cost is the Enhanced IC-PCP with Replication (EIPR) algorithm [18] in which Calheiros and Buyya use idle instances and budget surpluses to replicate tasks. Their experimental results show that the likelihood of meeting deadlines is increased by using task
replication. However, task replication in EIPR comes at an opportunity cost as the resources could be used for new rather than replicated computation.

In [27], Byun et al. presented the Partitioned Balanced Time Scheduling (PBTS) algorithm that estimates the minimum number of instances required to meet the deadline in order to minimize execution cost. The PBTS algorithm has three phases which are task selection, resource capacity estimation and the task scheduling phase. However, only one VM type is considered for provisioning and scheduling in order to simplify the estimation of resource capacity.

The Just in Time (JIT) algorithm proposed by Sahni and Vidyarthi in [13] is a dynamic cost minimization deadline constrained algorithm. The JIT algorithm attempts to combine pipeline tasks into a single task that can abrogate the data transfer time between co-located tasks. The majority of algorithms prioritize tasks to find the best candidate for execution however, no such policy is used in JIT.

2.3. Deadline and Budget Constrained Scheduling

The following three algorithms consider more than one constraint in scheduling workflows. In [28], Zheng and Sakellariou, introduce an extended version of HEFT algorithm with both budget and deadline constraints (BHEFT). BHEFT checks if a workflow can be scheduled based on the available budget and deadlines. To select the best possible instance in BHEFT, two variables named Spare Application Budget and Current Task Budget are used. This work while developed for cloud, used a grid cost model.

In [29], Poola et al. present a robust scheduling algorithm for heterogeneous cloud resources with time and cost constraints. Three resource selection policies are used: deadline, cost and robustness, and each user can prioritize the policies independently.

The scheduling algorithms proposed by Malawski et al. in [30] aim to maximize the number of serviced workflows while meeting given budget and deadline constraints. These scheduling algorithms are designed for workflows in an Infrastructure as a Service (IaaS) cloud. However, the authors consider only one instance type rather than the wide variety of types that are currently supported by commercial providers.
3. Problem Definition

3.1. Application Model

Workflows are the most widely used models for representing and managing complex distributed scientific computations [1]. A Directed Acyclic Graph (DAG) is the most common abstraction of a workflow. Using a DAG abstraction, a workflow is defined as a graph $G = (T, E)$ where $T = \{t_0, t_1, ..., t_n\}$ is a set of tasks represented by vertices and $E = \{e_{i,j} | t_i, t_j \in T\}$ is a set of directed edges denoting data or control dependencies between tasks. An edge $e_{i,j} \in E$ represents the precedence constraint as a directed arc between two tasks $t_i$ and $t_j$ where $t_i, t_j \in T$. The edge indicates that task $t_j$ can start only after completing the execution of task $t_i$ with all data received from $t_i$ and this implies that task $t_i$ is the parent of task $t_j$, and task $t_j$ is the successor or child of task $t_i$. Each task can have one or more parents or children. Task $t_i$ cannot start until all parents have completed.

3.2. System Model

In this article, we adopt the IaaS service model. The IaaS paradigm provides a service by offering instance types containing various amounts of CPU, memory, storage and network bandwidth at different prices. Workflows are executed on different instance types, and each instance type is associated with a set of resources.

We use a resource model based on the Amazon Elastic Compute cloud, where instances are provisioned on demand. The pricing model is a pay as you go with minimum hourly billing. Under this pricing model, if an instance is used for one minute, a user has to pay for the whole hour. We assume that cloud vendors provide access to unlimited number of instances and the instances are heterogeneous (denoted by $P = \{p_0, p_1, ..., p_h\}$, where $h$ is the index of the instance type). We also assume that all instances and storage services are located in the same region and also assume that the average bandwidth between the instances is essentially identical.

3.3. Definitions

Most studies on workflow scheduling assume that estimated execution time for workflow tasks are known. Task runtimes can be estimated using analytical modeling, empirical modeling and historical data. In this article,
we use scientific workflows generated using trace data from real applications [31]. Execution time (computation cost) for task $t_i$ on instance $p_j$ is denoted by $w_{t_i}^{p_j}$. All immediate predecessors of task $t_i$ are defined as:

$$\text{pred}(t_i) = \{ t_j \mid (t_j, t_i) \in T \}.$$  \hspace{1cm} (1)

Also, all immediate successors of task $t_i$ are defined as:

$$\text{succ}(t_i) = \{ t_j \mid (t_i, t_j) \in T \}.$$  \hspace{1cm} (2)

For example, predecessors of task $H$ are $D, E$ and $G$, successors of $C$ are $G$ and $F$.

A task without any parent is an entry task and a task without any children is called an exit task. In Figure 1, task A is an entry task and task J is an exit task. Thus, by definition we have:

$$\text{pred}(t_{\text{entry}}) = \{ \emptyset \},$$  \hspace{1cm} (3)

$$\text{succ}(t_{\text{exit}}) = \{ \emptyset \}.$$  \hspace{1cm} (4)
The completion time of a workflow is called the schedule length or makespan (denoted by $L_{ms}$). Because $t_{exit}$ is the last task that can be executed, the time until completing the exit task is defined as the makespan of a workflow.

\[ L_{ms} = FT(t_{exit}), \]  

(5)

where $FT(t_{exit})$ is the finish time of the last task in a workflow.

The amount of data transferred from task $t_i$ to task $t_j$ is called communication time (denoted by $C_{i,j}$), and this time is calculated as:

\[ C_{i,j} = \begin{cases} \frac{\text{data}}{\beta}, & p_i \neq p_j \\ 0, & p_i = p_j \end{cases} \]  

(6)

If task $t_i$ and task $t_j$ are executed on the same instance (denoted by $p_i \neq p_j$), data transfer between them is local and the communication cost is defined as zero. Otherwise, the communication cost is the ratio between the size of data (data) to be transferred from task $t_i$ to $t_j$ to the average bandwidth ($\beta$) in the data-center.

The Earliest Start Time (EST) of a task $t_i$ is calculated on the instance with the shortest execution time and defined as:

\[ EST(i) = \begin{cases} 0, & t_i = t_{entry} \\ \max_{t_j \in \text{pred}(t_i)} \{ EST(t_j) + w_{t_j} + C_{i,j} \}, & \text{Otherwise} \end{cases} \]  

(7)

where $w_{t_j}$ is the execution time of task $t_j$ on the fastest instance type.

The cost of executing task $t_i$ on instance $p_j$ is calculated as:

\[ TaskCost_{t_i}^{p_j} = \left\lceil \frac{w_{t_i}^{p_j}}{N_t} \right\rceil \times c_j, \]  

(8)

where $c_j$ is the cost of instance $p_j$ for one time interval and $N_t$ is the number of intervals. Finally, the overall cost of executing all tasks in a workflow is defined as:

\[ Cost_o = \sum_{t_i \in G} TaskCost_{t_i}^{p_j}. \]  

(9)

4. PDC and DCCP Algorithms

In this section, we present the extended deadline constrained algorithms, Proportional Deadline Constrained (PDC) and Deadline Constrained Critical
Path (DCCP). As a guide to this this section Figure 2 shows the sequence of steps for scheduling a workflow in both PDC and DCCP and indicates which sub-section details those parts of the algorithms.

Figure 2: Scheduling workflow with PDC and DCCP

4.1. Preprocessing Step
Both DCCP and PDC use a preprocessing step for partitioning tasks. In the preprocessing step, the tasks are partitioned into different levels based on their respective dependencies. Subsequently, the user-defined deadline $T_D$ is distributed over the levels established in the preprocessing step. Each level gets its own level deadline and all tasks in the same level have the same level-deadline.

4.1.1. Workflow Leveling
We aim to maximize task parallelism by partitioning tasks such that there are no dependencies between tasks in each level. Each level can therefore be thought of as a bag of tasks (BoT) containing a set of independent tasks.

There are two main algorithms for allocating tasks into different levels, Deadline Bottom Level (DBL) [32] and Deadline Top Level (DTL) [33]. Both DBL and DBT algorithms categorize tasks in bottom-top direction and top-bottom direction, respectively. In this article, we use the DBL algorithm to partition tasks over the different levels.
We describe the level of task $t_i$ as an integer representing the maximum number of edges in the paths from task $t_i$ to the exit task (see Fig. 1). The level number (denoted by $N_L$) associates a task to a BoT. For the exit task, the level number is always 1, and for the other tasks, it is determined by:

$$N_L(t_i) = \max_{t_j \in \text{succ}(t_i)} \{N_L(t_j) + 1\}, \quad (10)$$

where $\text{succ}(t_i)$ denotes the set of immediate successors of task $t_i$. All tasks are then grouped into Task Level Sets (TLS) based on their levels:

$$\text{TLS}(\ell) = \{t_i|N_L(t_i) = \ell\}, \quad (11)$$

where $\ell$ is an integer denoting the level in $[1 \ldots N_L(t_{entry})]$.

### 4.1.2. Proportional Deadline Distribution

Once all tasks are assigned to their respective levels, the tasks are proportionally distributed across each level based on the user deadline ($T_D$). Each sub-deadline assigned to a level is termed the level deadline ($T_{sd}(\ell)$). To meet the overall deadline, we attempt to ensure that every task in a level can complete its execution before the assigned sub-deadline. Firstly, the initial estimated deadline for each level ($\ell$) is calculated as:

$$\text{Initial}T_{sd}(\ell) = \max_{t_i \in \text{TLS}(\ell)} \{\text{ECT}(t_i)\}, \quad (12)$$

where $\text{ECT}(t_i)$ denotes the Earliest Completion Time (ECT) of task $t_i$ over all instances and the ECT is defined as

$$\text{ECT}(t_i) = \max_{\ell \in \text{pred}(t_i)} \{\text{Initial}T_{sd}(\ell), \text{EST}(t_i)\} + w_{t_i}, \quad (13)$$

where $\text{EST}(t_i)$ is defined in equation 7, $\text{pred}(t_i)$ denotes the set of predecessors of task $t_i$; $w_{t_i}$ denotes the minimum execution duration for task $t_i$ and $\ell$ indicates the parent level $t_i$. The task, $t_{entry}$ has no predecessors, its ECT is equal to zero. In equation 12 the maximum ECT of all tasks in a level is used as the overall estimate for that level. This duration is effectively the absolute minimum time that is required for all tasks in a level to complete execution in parallel.
After calculating the estimated deadline value for all levels, we distribute the user deadline among all tasks non-uniformly based on a deadline proportion denoted by $\propto_{\text{deadline}}$ in equation (14):

$$\propto_{\text{deadline}} = \frac{T_D - InitialT_{sd}(1)}{InitialT_{sd}(1)},$$

where $InitialT_{sd}(1)$ is the level that contains the exit task.

We then compute length of each level deadline as a function of this deadline proportion to each level as follows:

$$T_{sd}(\ell) = InitialT_{sd}(\ell) + (\propto_{\text{deadline}} \times |InitialT_{sd}(\ell)|).$$

Intuitively, the levels with longer executing tasks gain a larger share of the user deadline.

4.2. Task Prioritization

4.2.1. PDC Algorithm (A Single task)

In each step of the PDC algorithm, tasks that are ready to execute are placed into the task ready list. A task is ready when all of its parents have been executed and all its required data is readily accessible. To select a task, at first all tasks in the ready list should first be prioritized. We used eight different policies in order to show how the order of execution can influence the scheduling results, particularly the cost. These are summarized in table 1.

We will discuss these result later (see section 5.2).

1. Upward Rank ($rank_u$): This ranking is presented in [21]. The upward rank is the length of critical path from task $t_i$ to task $t_{exit}$ and is calculated by equation (16), where $\overline{w_i}$ and $\overline{c_{ij}}$ are the average execution time and average communication time of task $t_i$, respectively. The rank is called an upward rank because the ranking process starts from the exit node and ranks are calculated recursively by traversing the DAG to the entry node.

2. Downward Rank ($rank_d$): The downward rank [21] starts from the entry node and is computed recursively by traversing the DAG to the exit node. $rank_d(t_i)$ is the longest distance from $t_{entry}$ to task $t_i$, excluding the computation cost of task itself, where $rank_u(t_i)$ is the length of the critical path from task $t_i$ to $t_{exit}$, including the computation cost of the task itself [21].
<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
<th>Formula</th>
<th>Policy Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upward Rank ((rank_u))</td>
<td>The length of critical path from task (t_i) to task (t_{exit})</td>
<td>(rank_u(t_i) = w_i + \max_{t_j \in succ(t_i)} (\overline{c}_{i,j} + rank_u(t_j))) (16)</td>
<td>static</td>
</tr>
<tr>
<td>Downward Rank ((rank_d))</td>
<td>Starts from the (t_{entry}) and is computed recursively by traversing the DAG to (t_{exit})</td>
<td>(rank_d(t_i) = \max_{t_j \in pred(t_i)} (w_j + \overline{c}_{j,i} + rank_d(t_j))) (17)</td>
<td>static</td>
</tr>
<tr>
<td>Sum Rank ((rank_s))</td>
<td>Sum of the upward and downward rank</td>
<td>(rank_s(t_i) = rank_u(t_i) + rank_d(t_i))</td>
<td>static</td>
</tr>
<tr>
<td>Minimum Execution Time (MinExe)</td>
<td>Lowest execution time is given first priority</td>
<td>min((w_i))</td>
<td>static</td>
</tr>
<tr>
<td>Maximum Execution Time (MaxExe)</td>
<td>Tasks with a longer execution time has higher priority</td>
<td>max((w_i))</td>
<td>static</td>
</tr>
<tr>
<td>Random</td>
<td>Tasks are picked from the ready list at random</td>
<td>—</td>
<td>static</td>
</tr>
<tr>
<td>Earliest Completion Time (ECT)</td>
<td>The task that finishes first will be the best candidate for execution</td>
<td>min((FTw_i))</td>
<td>dynamic</td>
</tr>
<tr>
<td>Earliest Deadline First (EDF)</td>
<td>Tasks with minimum EDF have highest priority</td>
<td>(EDF(t_i) = Level_{deadline}^{t_i} - EST(t_i))</td>
<td>dynamic</td>
</tr>
</tbody>
</table>
3. Sum Rank($rank_s$): This rank gives equal importance to both the up-rank and downrank and is calculated as the arithmetic sum of $rank_s$ and $rank_d$.

4. Minimum Execution Time (MinExe): For each task in the ready list, the minimum execution time on all VMs types is calculated and the task with the lowest execution time is given first priority.

5. Maximum Execution Time (MaxExe): Similar in principle to the minimum execution time, with the only difference being that the task with a longer execution time has higher priority.

6. Random: In this policy, tasks are picked from the ready list at random.

7. Earliest Completion Time (ECT): For each task the earliest completion time on all VMs launched is calculated. The task that finishes first will be the best candidate for execution.

8. Earliest Deadline First (EDF): Tasks with a minimum EDF have highest priority among all ready tasks.

4.2.2. DCCP Algorithm (Multiple tasks: CCP definition)

A Critical Path (CP) is the longest path from the entry to exit node of a task graph [34]. The length of critical path ($|CP|$) is calculated as the sum of computation costs and communication costs, and can be considered as the lower bound for scheduling a workflow.

Several heuristics that utilize critical paths have been proposed for addressing the workflow scheduling problem [21, 34, 35]. The set of tasks containing only the tasks ready for scheduling constitutes a constrained critical path (CCP) [36]. In the DCCP algorithm, the CCP in a workflow is determined based on HEFT upward rank and downward rank [21], then we apply a set of new ranking methods defined as follows:

**modified upward rank** :

$$M_{rank_u}(t_i) = \overline{w_i} + \sum_{t_j \in succ(t_i)} (\overline{c_{i,j}}) + \max_{t_j \in succ(t_i)} (rank_u(t_j)) \quad (18)$$

**modified downward rank** :

$$M_{rank_d}(t_i) = \sum_{t_k \in pred(t_i)} (\overline{c_{k,i}}) + \max_{t_k \in pred(t_i)} (\overline{w_k} + rank_d(t_k)) \quad (19)$$

The difference between our modified rank from standard rank is that the modified rank aggregates a task’s predecessors’ or successors’ communication
time instead of selecting the maximum. With the modified rank, tasks with higher out-degree or in-degree have higher priorities. As a result, these tasks are executed first with higher probability and more tasks on the next CCP can be considered as ready tasks.

In this article, we use the sum rank to find all CPs [21]:

\[ rank_s = rank_u + rank_d \]  

(20)

In DCCP, all tasks are first sorted based on their \( rank_{sum} \) values and those tasks with the highest values are selected as the first CP. All tasks in the first CP are labeled as visited tasks. Proceeding in the same way, all CPs in a workflow can be found.

4.2.3. An illustrative example

We consider a sample DAG that contains 11 tasks as shown in Figure 3. The numbers associated with each edge shows the data transfer time between tasks. The data could either be direct or indirect via shared storage. For this article the only difference is in the absolute time required, and for simulation we only consider direct transfers. The average execution time (\( \bar{w} \)) of each task is displayed in Table 3.

1. A single task: Upon executing task 0, all its children are ready for execution. The different start time, end time and data transfer time (blue intervals) are shown in Figure 4. Different policies select different task (Table 2).

<table>
<thead>
<tr>
<th>Selected Task</th>
<th>MinExe</th>
<th>MaxExe</th>
<th>ECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_2 )</td>
<td>( t_3 )</td>
<td>( t_1 )</td>
<td></td>
</tr>
<tr>
<td>( t_1 )</td>
<td>( t_1 )</td>
<td>( t_1 )</td>
<td></td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
<td></td>
</tr>
</tbody>
</table>

2. Multiple tasks: The first CP is obtained based on the highest sum rank which is the aggregation of \( rank_u \) and \( rank_d \) and this yields \((0 \rightarrow 1 \rightarrow 4 \rightarrow 9 \rightarrow 11)\). Regardless of any previously selected tasks, proceeding in the same way, other CPs are found as displayed in Table 4. The next step is traversal of CPs to find CCPs in a round-robin order. The first CCP consists of \((0 \rightarrow 1)\) as other tasks in the first CP are not yet ready. For example, consider \( t_4 \) which is in the first CP, this task cannot be added to the CCP as one of its parents, \( t_2 \), has
Figure 3: A sample DAG with 11 tasks

Figure 4: Ready tasks and rank values (shown within each bar) after execution of task 0
not yet been added to any CCPs. As no ready tasks can be found in the first CP, a second (new) CP is constructed. In the new CP we have $t_2$ which is a ready task, as its only parent has already been included in a previous CCP. Thus, the second CCP consists of three tasks ($2 \rightarrow 5 \rightarrow 8$) having excluded $t_{10}$ from the second CP. Similarly, other CCPs are generated by using the remaining CPs. The different CCPs calculated by our modified rank approach are presented in Table 5.

### Table 3: Ranks values

<table>
<thead>
<tr>
<th>Task</th>
<th>$\omega_i$</th>
<th>$\text{rank}_u$</th>
<th>$\text{rank}_d$</th>
<th>$\text{rank}_s$</th>
<th>$\text{rank}_u$</th>
<th>$\text{rank}_d$</th>
<th>$\text{rank}_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22</td>
<td>190</td>
<td>0</td>
<td>190</td>
<td>284</td>
<td>0</td>
<td>284</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>142</td>
<td>48</td>
<td>190</td>
<td>142</td>
<td>48</td>
<td>190</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>150</td>
<td>38</td>
<td>188</td>
<td>203</td>
<td>38</td>
<td>241</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>96</td>
<td>39</td>
<td>135</td>
<td>96</td>
<td>39</td>
<td>135</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>84</td>
<td>106</td>
<td>190</td>
<td>84</td>
<td>115</td>
<td>199</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>110</td>
<td>78</td>
<td>188</td>
<td>140</td>
<td>78</td>
<td>218</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>65</td>
<td>74</td>
<td>139</td>
<td>65</td>
<td>85</td>
<td>150</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>70</td>
<td>115</td>
<td>185</td>
<td>89</td>
<td>115</td>
<td>204</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>75</td>
<td>113</td>
<td>188</td>
<td>75</td>
<td>113</td>
<td>188</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>41</td>
<td>149</td>
<td>190</td>
<td>41</td>
<td>173</td>
<td>214</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
<td>44</td>
<td>144</td>
<td>188</td>
<td>44</td>
<td>156</td>
<td>200</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
<td>180</td>
<td>190</td>
<td>10</td>
<td>236</td>
<td>246</td>
</tr>
</tbody>
</table>

### 4.3. Instance Selection in PDC

At the point the algorithms perform instance selection: (i) each task is already assigned a level, (ii) the deadline for each level is already determined, and (iii) the priority of each ready task is already assigned. During instance selection, a trade off must be made between execution time and cost. To demonstrate this trade off, we show the expressions for both the time and the cost of executing each task on each instance type in equations (21) and (22), forming two sets of expressions for Time and Cost.

Firstly, the time needed for the current task, $t_i$, on the instance $p_j$ is calculated by $\text{ECT} (t_i, p_j)$. The ECT is the earliest time that a task can finish on an instance which is defined earlier in equation (13). Using this observation, we can then compute how much the estimated level deadline
Table 4: CPs and CCPs based on standard ranks

<table>
<thead>
<tr>
<th>Critical Path</th>
<th>Constrained Critical Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→1→4→9→11</td>
<td>0→1</td>
</tr>
<tr>
<td>2→5→8→10</td>
<td>2→5→8</td>
</tr>
<tr>
<td>3→6</td>
<td>3→6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4→9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: CPs and CCPs based on modified ranks

<table>
<thead>
<tr>
<th>Critical Path</th>
<th>Constrained Critical Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→2→5→7→11</td>
<td>0→2→5→7</td>
</tr>
<tr>
<td>1→4→9</td>
<td>1→4→9</td>
</tr>
<tr>
<td>3→6→10</td>
<td>3→6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

of the current task differs from the earliest completion time of task on the instance $p_j$:

$$
\text{Time}_{p_j}^{t_i} = \frac{T_{sd}(N_L(t_i)) - \text{ECT}(t_i, p_j)}{T_{sd}(N_L(t_i)) - \text{ECT}(t_i)}
$$

In equation (21), $T_{sd}(\cdot)$ is the deadline that is assigned to the level which contains the current task. Also, $\text{ECT}(t_i)$ is the minimum execution time among all instances that keeps our current task on schedule.

The values of Time for task $t_i$ are related to instance types, wherein the lower value of Time means running on a cheaper instance. The reason is that the values of $\text{ECT}(t_i, p_j)$ is bigger on an instance with a lower processing capacity. Also, if the value of Time is negative, it means that the current task on the selected instance will exceed the level deadline i.e. $\text{ECT}(t_i, p_j) >$
In the expression for Cost, given earlier in $8$, $TaskCost_i$ refers to the cost of scheduling the current task $t_i$ on instance $p_j$. In equation (22), the worst cost (maximum cost) and best cost (minimum cost) of executing the task $t_i$ among all instances are $TaskCost_{worst}$ and $TaskCost_{best}$, respectively.

$$Cost_{p_j \ t_i} = \frac{TaskCost_{worst} - TaskCost_i}{TaskCost_{worst} - TaskCost_{best}}$$

(22)

To find the best instance, we use a Cost Time Trade-off Factor (CTTF) in equation (23) that considers a trade-off between cost and time.

$$CTTF_{p_j \ t_i} = \frac{Cost_{p_j \ t_i}}{Time_{p_j \ t_i}}$$

(23)

When an instance is first provisioned, the instance is billed on an hourly interval until it is terminated. Therefore the first task assigned to an instance in a particular billing interval incurs the entire cost of that interval. As a consequence, if other tasks can be executed during that paid interval, then there is no additional execution cost for executing them. Therefore, during instance selection, we first prioritize the reuse of such instances (i.e. when Cost in equation (22) is 1), provided that the level deadline is not exceeded (i.e. when Time in equation (21) is positive).

If there are more than one paid instances, the PDC selects the instance with the minimum execution time (faster instances). If no such instances are available, it will attempt to use a provisioned but as yet unused instance, or as a last resort it creates a new instance.

### 4.4. Instance Selection in DCCP

In the Instance Selection phase, the DCCP algorithm identifies the most appropriate instance to execute CCPs. All tasks in a CCP are executed on the same instance to minimize communication cost between them. The time needed for the current CCP (denoted by $(CCP_i)$), to execute on the instance $p_j$ is calculated by $ECT(CCP_i, p_j)$. Work in scheduling generally assumes such an estimate can be calculated. In practice this is difficult, however work is underway to profile workflow tools and underlying cloud systems to provide usable estimates for use in production systems [37].

The ECT is the earliest time that a CCP can complete execution on an instance (as defined in equation (13) for a single task). The differences
between the estimated level deadline and earliest completion time of the current CCP on the instance \( p_j \) is determined by:

\[
\text{Time}_{\text{CCP},p} = T_{sd} (N_L (t_i)) - \text{ECT} (\text{CCP},p_j)
\]  

where \( T_{sd} \) is the deadline that is assigned to the level (given by \( N_L (\cdot) \)) which contains the last task \( t_i \) on the current CCP. There is a possibility that this value may be negative which means the current CCP exceeds the level deadline (\( \text{ECT} (\text{CCP},p_j) > T_{sd} (N_L (t_i)) \)). The cost of executing all tasks on current CCP on instance \( p_j \) is denoted by \( \text{Cost}_{\text{CCP},p_j} \):

\[
\text{Cost}_{\text{CCP},p_j} = \sum_{t_i \in \text{CCP}_i} \text{TaskCost}_{p_j, t_i}
\]  

Three different scenarios to find the most appropriate instance can be considered:

1. Most cloud providers, such as Amazon, charge based on 60 minutes interval. When a task is scheduled on an instance, the whole billing interval is charged no matter how much the instance is used. Therefore, if other tasks can be executed on the same VM during that paid interval, their execution cost is zero. To find the best instance in DCCP, the priority is to select an instance with residuals to execute a CCP. This is subject to its earliest completion time does not exceed the level deadline. The instance with minimum ECT is selected (the fastest one).

2. A new instance is provisioned if no instances could be found in the previous step. For example, at the beginning of the scheduling to assign the first CCP, an instance should be provisioned as there is no paid instances. For this purpose, DCCP searches among instances that can meet the level deadline and select the cheapest one.

3. In tight deadlines, there is a possibility that none of the instances can meet the task level’s sub-deadline (i.e., when \( \text{Time}_{\text{CCP},p} \) is negative). If this condition for a CCP is met, it does not mean that its impossible to meet the overall user defined deadline. Rather, it means that the sub-deadline will be violated. In this case we select the best available instance - as overall the schedule may still be met.
4.4.1. Backfilling in DCCP

Scheduling of a workflow that consists of dependent tasks creates resource utilization gaps between the execution of tasks. The principal reason is that tasks must wait for data from its parents. Therefore, there are idle time slots formed between scheduled tasks on each resource. Moreover, the utilization of cloud resources depends on how tasks are placed together. Instance fragmentation and resource wasting occurs if tasks are not packed firmly. Scheduling algorithms can consider these time slots for executing ready tasks on different resources. Consequently, filling up the idle slots decreases the makespan and maximizes the overall instance utilization.

While backfilling policies are widely used to reduce fragmentation, this has not been done in previously in workflow scheduling. To our knowledge the use of backfilling strategies in DCCP is unique. In this section, we show that backfilling increases instance utilization and this improved utilization leads to cost saving. In PDC, we use a cost-time trade off for instance selection, this approach narrows our choices of instances for backfilling. Therefore, the backfill algorithm is only used in conjunction with DCCP. Three different policies are considered which exploit such idle slots to efficiently schedule tasks which are First Fit (FF), Best Fit (BF) and Worst Fit (WF). Each CCP can be placed in a residual according to one of the following policies:

1. First Fit (FF): a CCP can be inserted into the first gap where it fits.
2. Best Fit (BF): a CCP is placed into the schedule gap where it leaves the minimum sized residuals.
3. Worst Fit (WF): a CCP is inserted into the schedule gap where it leaves the maximum sized residuals.

In Section 5.3 we discuss how using of backfill policies significantly improves overall utilization.

4.5. Time Complexity

The time complexity of the two proposed algorithms is an important metric for benchmarking different scheduling algorithms. Consider a workflow represented by a DAG $G = (T, E)$ with $n$ tasks. If we assume that a DAG is fully connected, the maximum number of dependencies between tasks is $(n)(n - 1)/2$. Processing of all tasks and its dependencies requires a time complexity of $O(n^2)$. Besides, processing task dependencies, other computations in PDC that must be taken into account are the task selection phase and the instance selection phase, which are distinct from each other.
To compute the time complexity of task selection phase, all ready tasks \((n)\) should be examined on all available processors \((p)\) that need computation of \(O(np)\). Similarly, to select all workflow tasks, the time complexity of task selection is \(O(n^2p)\). In the resource selection phase, selected tasks are evaluated on all available instances with complexity of \(O(p)\). Thus, the total time complexity of resource selection is \(O(np)\). The total time for PDC is \(O(n^2 + np + np)\), where the algorithm complexity is of the order \(O(n^2p)\). The only difference between DCCP with PDC is in calculating constrained critical paths. For this purpose, the calculation of upward and downward rank occurs with time complexity of \(O(n^2p)\). Therefore, the DCCP algorithm is also of the order of \(O(n^2p)\).

5. Evaluation

In this section we compare the performance of the PDC and DCCP algorithms, with the well known IC-PCP [12] algorithm and the recently published JIT [13] algorithm. We use a simulator to compare the performance of all four algorithms. Simulations are well accepted as the first approach to evaluate new techniques for workflow scheduling problem. It allows researchers to test the performance of newly developed algorithms under a controller setting. For this purpose, all four algorithms are implemented and evaluated in CloudSim [11].

Our simulation scenario is configured as a single data-center and six different instance types. The characteristics of the instances are based on the Amazon EC2 instance configurations presented in Table 6.

The average bandwidth between instances is fixed to 20 MBps, based on the average bandwidth published by AWS [38]. The processing capacity of an EC2 unit is estimated at one Million Floating Point Operations Per Second (MFLOPS) [39]. The estimated execution times are scaled by instance type CPU performance. In an ideal cloud environment, there is no provisioning delay in resource allocation. However, some factors such as the time of day, operating system, instance type, location of the data center, and number of requested resources at the same time, can cause delays in startup time [40]. Therefore, in our simulation, we adopted a 97-second boot time based on previous measurements of EC2 [40].

In order to evaluate the performance of our algorithms with a realistic load, we use five common scientific workflows: Cybershake, Epigenomics, Montage, LIGO and SIPHT. The characteristics and task composition of
Table 6: Instance Types based on Amazon EC2

<table>
<thead>
<tr>
<th>Type</th>
<th>ECU</th>
<th>Memory(GB)</th>
<th>Cost($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>3</td>
<td>3.75</td>
<td>0.067</td>
</tr>
<tr>
<td>m4.large</td>
<td>6.5</td>
<td>8</td>
<td>0.126</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>13</td>
<td>15</td>
<td>0.266</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>26</td>
<td>32</td>
<td>0.504</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>53.5</td>
<td>64</td>
<td>1.008</td>
</tr>
<tr>
<td>m4.10xlarge</td>
<td>124.5</td>
<td>160</td>
<td>2.520</td>
</tr>
</tbody>
</table>

these workflows have been analyzed in published works cited in the related work section [31, 41]. To evaluate the performance of these algorithms, we choose different deadlines chosen from tight to relaxed. Additionally, we calculate the fastest schedule (denoted by $FS$) as a baseline schedule. Effectively, this baseline is the fastest possible execution - ignoring costs.

$$FS = \sum_{t_i \in CP} (w_j^i)$$  \hspace{1cm} (26)

where $w_j^i$ is the computation cost of task $t_i$ on the fastest instance $p_j$.

We define the deadline as a function of the fastest schedule and this deadline is expressed in equation [27] in which the deadline varies from tight to moderate to relaxed:

$$\text{deadline} = \alpha \times FS, \quad 0 < \alpha < 20.$$  \hspace{1cm} (27)

The deadline factor $\alpha$ starts from 1 to consider very tight deadlines (typically approaches the fastest schedule) and is increased by one up to a value of 20, which results in a very relaxed deadline.

The Amazon EC2 instances charge on an hourly interval from the time of provisioning. We configure our simulator to reflect this charging model and we use a time interval of 60 minutes in our simulations. To compare performance with respect to different workflow sizes we evaluated workflows with 50, 100, 200, 500 and 1000 tasks. However, as these results did not vary significantly we present here only workflows with 1000 tasks.
We used the Pegasus workflow generator \cite{31} to create representative workflows with the same structure as five real world scientific workflows (Cybershake, Epigenomics, Montage, LIGO and SIPHT). For each workflow structure, and each deadline factor, 100 distinct Pegasus generated workflows were scheduled in CloudSIM and the performance of the scheduling algorithms are detailed in the following section.

5.1. Performance Metrics

To evaluate the algorithms under test, we elected to use the following performance metrics: Success Rate (SR), Normalized Schedule Cost (NSC) and Throughput.

- **Success Rate (SR):** success rate of each algorithm (SR), calculated as the ratio between the number of simulation runs that successfully met the scheduling deadline and the total number of simulation runs (denoted by \( n_{Tot} \)), defined as:

  \[
  SR = \frac{n(k)}{n_{Tot}},
  \]

  where \( n(k) \) is the cardinality of the set \( k \) and \( n_{Tot} = 100 \).

- **Normalized Schedule Cost (NSC):** To compare the monetary cost between the algorithms, we consider the cost of failure in meeting a deadline. For this purpose, a weight is assigned to average cost returned by each algorithm. Let \( k \) denote the set of a simulation runs that successfully meets the scheduling deadline, thus the weighted cost is calculated as:

  \[
  Cost_w = \sum_{k} \frac{Cost_o(k)}{SR},
  \]

  where \( Cost_o(k) \) was defined earlier in equation 9 and it is the cost for the experiments that meet the deadline. Thus, the NSC is defined as:

  \[
  Cost_{ns} = \frac{Cost_w}{\text{min}_C},
  \]

  We consider the cheapest schedule (denoted by \( \text{min}_C \)) as scheduling of all tasks on the cheapest instance according to their precedence constraints.

- **Throughput:** is the amount of work that can be done in a given deadline interval by each algorithm. Million Floating Point Operations Per Second (MFLOPS) is used as a measure of the throughput.
5.2. Task Selection in PDC

The task selection step is a characteristic of all list based scheduling algorithms. In this section we evaluate task selection in PDC using a set of eight different ranking policies in order to evaluate the importance of and sensitivity to ranking. We previously evaluated the task selection step for DCCP in [9] where two algorithms for constructing the constrained critical path were evaluated. Figures 5 to 9 show the results of different task selection policies in PDC as defined in Section 4.2.1.

Figure 5: Task selection results for Montage

Workflows differ remarkably in their characteristics including structure, size, computation and communication requirements. Each workflow is constructed of various components including process, pipeline, data distribution, data aggregation and data redistribution [31]. The size of scientific workflows varies from small number of tasks taking a few minutes to execute, to millions of tasks that require days to execute. Moreover, workflows also differ in terms of data transfer operations, example of such transfers are fetching input data, moving intermediate data generated within a workflow and output data. For each workflow structure, and each deadline factor, 100 distinct Pegasus generated workflows are simulated using CloudSim.
Figure 6: Task selection results for SIPHT

Figure 7: Task selection results for LIGO
Figure 8: Task selection results for Cybershake

Figure 9: Task selection results for Epigenomics
The average execution time and average communication time are used for task ranking by Upward, Downward and Sum rank. Accordingly, the impact of instances that are launched during scheduling are not accounted for by these policies. We call these static policies, as they do not change when additional instances are launched by the scheduler.

The Earliest Completion Time (ECT) and Earliest Deadline First (EDF) polices require continual recomputation as execution of a task on a VM leads to changes in ECT and EST for all other tasks on that VM. We call these dynamic policies, as they change when additional instances are launched by the scheduler.

The dynamic ranking policies performed best on the Montage (Figure 5) and Cybershake (Figure 5) workflows, whereas the results were largely ranking agnostic in LIGO (Figure 7) and Epigenomics (Figure 9). The most interesting result was the SIPHT workflow (Figure 6), where the results were largely unpredictable. None-the-less, overall the dynamic EDF policy produced the lowest costs over all workflows tested.

The results of this set of experiments suggest that the structure of workflows can significantly impact the ranking and scheduling cost. We note that the workflows where the policies performed most consistently had a high degree of structural and runtime symmetry – where each task in such a sequence usually has the same amount of data as input, and in turn generates and distributes equal information as output to their children. Indeed, the workflows for which the policy was agnostic suggest support for this conjecture. The most unpredictable workflow, SIPHT was strongly asymmetric in both structure and runtime. The structures of these workflows can be found in [31, 41].

In future work, we will investigate the impact of workflow structure – where we will look for a measure of symmetry and consider how this can be incorporated into scheduling decisions. Findings in this area may also lead to better practice in the design of workflows themselves.

Although cost differences may seem negligible between some of the policies, in multiple of datasets the variance could be significant. This shows that the task selection order could play a key role in minimizing the cost.

5.3. Backfilling in DCCP

In this section, we evaluate backfilling strategies in terms of the number of provisioned instances. A more effective backfilling strategy will ultimately
provision fewer instances and thereby reduce cost. The three different strategies (FF, BF and WF) outlined in Section 4.4.1 are used. We also limit the simulation to a single instance type (type 2 in Table 6) to ensure that the comparison across the algorithms is fair. Two data intensive workflows, LIGO and Montage, are chosen for evaluation with three different deadline intervals. In Figures 10 and 11, graphs in left columns show the instance utilization based on the VM creation order the right column shows the same sorted by utilization. The X-axis is the total number of instances and the Y-axis is the utilization rate.

The worst fit policy has the best performance because it launches fewer instances, and the number of launched instances has a direct effect on cost. Worst fit reduces further fragmentation of the residuals leaving larger allocatable blocks. A small set of high utilized VMs leads to lower overall cost in worst fit policy compared to further low utilized VMs in other policies. Overall, the number of instances decreases as the deadline is relaxed. In a case of the moderate and relaxed deadlines in LIGO, interval 12 and 19, it is observed that worst fit needs almost the half number of VMs. The same observation is true for MONTAGE with interval 19, worst fit approximately requires one-third of VM numbers compared to others. Considering benefits of cost saving in worst fit, we used this policy in DCCP algorithms for cost comparison analysis in Section 5.4.

5.4. Cost Comparison Analysis

Ultimately the cost vs. deadline performance of each algorithm is the most significant basis for evaluating their performance. In this section we evaluate each of the algorithms using six different instance types with different characteristics as described in Table 6. As expected, experimental results in Figure 12 show that the cost of the workflow scheduling generally decreases as the deadline factor increases.

In most cases, the PDC and DCCP algorithms outperform both IC-PCP and JIT, achieving the lowest overall cost over all workflows and deadlines. Like all heuristics, there are points at which their performance is not as good, but these are in the minority, and for small values. For example, in Cybershake, the cost of PDC at most deadlines is approximately 10% that of the cost incurred by the worst performer, JIT.

Another interesting result is from the Epigenomics workflow, where while IC-PCP achieves the lowest costs with relaxed deadlines (12 → 20), this
Figure 10: VM utilization for three different deadline intervals with Backfilling policies for LIGO.
Figure 11: VM utilization for three different deadline intervals with backfilling policies for MONTAGE.
Figure 12: Normalized Cost vs. deadline for five different datasets.
algorithm also is unable to generate any viable schedule, at any cost, for the majority of the tighter deadlines.

As a final observation, from Figure 12, the cost of finding a schedule when deadline is tight for Montage and Cybershake is extremely high. It can be explained by considering the structure of these workflows. For example in Montage, more than 800 parallel tasks out of 1000 in first two levels need to be scheduled. Therefore, when deadline is tight, all algorithms need to lease many instances in parallel to finish elementary tasks that makes the schedule cost very expensive

5.5. Success Rate Analysis

Figure 13 shows the relative Success Rate (SR) of each algorithm as the deadline factor, $\alpha$, is increased from 1 to 20.

A low success rate indicates that the algorithm cannot find a makespan that meets the deadline (in most datasets). The best overall performers are the PDC and JIT algorithms, which exhibit a success rate of 100% for most deadlines. We observe that the relaxing of the deadline causes the success rate of each algorithm to increase except for IC-PCP. Although we expected to have fewer failures when the deadline is relaxed, the behavior of IC-PCP in different intervals is contrary to these expectations. The highest failure occurs when deadline factor, $\alpha < 4$, with 100% failure, except in Cybershake. IC-PCP also has the worst performance in Epigenomics below a deadline factor of 16, IC-PCP can find a schedule for less than 2% of the datasets before its deadline is reached.

The best performance of IC-PCP belongs to Cybershake, which has a success rate of above 60% in all intervals. The DCCP algorithm in all scientific workflows has 100% success when deadline is more relaxed. The maximum failure in DCCP happens in Epigenomics when deadline is tight. No significant differences were found between the PDC and JIT whereas both are able to finish workflows for more than 95% of the deadlines. Although in most of the tested deadlines JIT can find a solution, JIT generates expensive schedules as it is discussed in 5.4.

5.6. Throughput Analysis

The throughput of each algorithm is displayed in Figure 14. The X-axis in Figure 14 is the cost based on the deadline intervals. The Y-axis is the number of MFLOPS in billion.
Figure 13: Success Rate for five different datasets.

(a) Montage

(b) SIPHT

(c) LIGO

(d) Cybershake

(e) Epigenomics
Figure 14: Throughput for five different datasets.
In Figure 14, the top left corner indicates better performance at a lower cost. Clearly the throughput is dependent on the success rate – and therefore over all intervals and workflows the throughput of the best algorithms, PDC and JIT are essentially equal. However, the cost difference between PDC and JIT is significant as shown in the graph. The best performance of DCCP is in MONTAGE and LIGO (both are data intensive workflows), in which DCCP is close to PDC with a similar cost. IC-PCP has the worst performance in almost all workflows, which is directly related to the low success rate of this algorithm.

6. Conclusion

In this article we address the problem of scientific workflow scheduling in dynamically provisioned commercial cloud environments. The ever increasing use of cloud computing by scientists has highlighted opportunities for improving utilization of cloud infrastructure by improving response time while decreasing the net cost of computation. In this article we employ workflow scheduling to achieve lower cost with good response times in cloud environment. Workflow scheduling in cloud environment differs from grid and cluster computing environments primarily in the elastic resource provisioning and pay-per-use charging model. Therefore, workflow scheduling in clouds requires a different approach in mapping tasks to resources while limiting the cost.

To address this we introduced new algorithms, Proportional Deadline Constrained (PDC) and Deadline Constrained Critical Path (DCCP). PDC operates by maximising the parallelism in a workflow by separating it into logical levels and then proportionally sub dividing the overall workflow deadline over them. The DCCP algorithm is similar to PDC, the main difference is that it also determines the constrained critical path through the workflow in order to co-locate tasks that communicate on the same instance. We then evaluated these algorithms (via CloudSim simulation) against two previously published algorithms (IC-PCP and JIT), using a variety of metrics – including success rate, normalised cost, and throughput. We also investigated the influence of the task selection step in the PDC algorithm and backfilling strategies in DCCP. The simulations were conducted using five scientific workflows, Montage, SIPHT, LIGO, Cybershake and Epigenomics, and these were generated by the Pegasus workflow generator.
In terms of cost performance, overall the PDC and DCCP algorithms returned the lowest compute cost, over all workflows and instance configurations. Of particular note, for the Cybershake workflow, PDC returned costs approximately 10% of those incurred by JIT. In terms of success rate and throughput, the best overall performers were the PDC and JIT algorithms, although the JIT algorithm was many times more expensive in terms of cost. We also investigated the effect of eight different policies to evaluate task selection order on scheduling performance in PDC. In doing so, we appear to have found an interesting connection between the symmetry of a workflow and the performance of the scheduling algorithms, which is worthy of further investigation.

For DCCP we strategically backfill residuals in provisioned instances. This approach is most effective in data intensive workflows due to the reduction in data movement. Worst fit resulted in higher utilisation on fewer instances than either first fit or best fit. Worst fit reduces further fragmentation of the residuals by leaving larger allocatable blocks.

In this article we present two algorithms for scheduling workflows on dynamically provisioned elastic cloud resources. Overall, both our algorithms are able to achieve a consistently high success rates and throughput, while in most cases presenting the lowest overall pay-per-use cost. In general DCCP slightly outperforms PDC as most workflows we tested are data intensive.


doi:http://dx.doi.org/10.1016/j.future.2015.01.004


[37] R. Chard, K. Bubendorfer, B. Ng, Network health and e-science in commercial clouds, Accepted: Future Generation Computer Systems doi: 10.1016/j.future.2015.06.001

