Deadline-Constrained Big Data Workflow Scheduling in the Cloud: the LPOD Algorithm

Changxin Bai, *Student-Member, IEEE,* Dr Shiyong Lu, *Senior Member, IEEE,* Dr Dunren Che, *Senior Member, IEEE* and Junwen Liu, *Student-Member, IEEE*

Abstract—Workflow scheduling in the cloud has been an important and challenging problem. It is challenging because one needs to consider not only the characteristics of the underlying cloud platform, including the elastic resource provisioning model and the pay-as-you-go price model, but also QoS requirements, such as monetary budget constraints and deadline constraints. In this paper, we focus on the deadline-constrained workflow scheduling problem in the cloud. Meta-heuristic based algorithms takes longer response time with elaborate parameter tuning process. In light of these, a path based scheduling algorithm using an innovative dynamic programming strategy, named LPOD, is proposed to find optimized workflow scheduling solutions for a given deadline-constrained workflow in a cloud computing environment. LPOD has been successfully integrated into the workflow executor of DATAVIEW (a popular open-source big data workflow management system). One salient feature of LPOD is that it collects and uses provenance information (e.g., task execution time and data transfer time) to improve workflow scheduling solutions. Several case studies have been carefully conducted using both synthetic and real-life workflows in DATAVIEW. The experimental results show that the proposed algorithm is efficient and can scale to many tasks and large datasets, demonstrating superior workflow scheduling quality and performance than the state-of-the-art path based algorithm IC-PCP and the untuned meta-heuristic algorithm GA.

Index Terms—Workflow Scheduling Algorithm, Deadline Constrained Scheduling, Optimization, Workflow Execution, Cloud Computing, Big Data.

1 Introduction

Workflow, as a parallel high-performance computing model, has been broadly studied and used to orchestrate different computation tasks involving complex data dependencies in various research domains such as astronomy [1], biology [2] [3], seismology [4], medical care [5] [6], and big data analytics [7]. Those applications must be handled by modern workflow management systems such as DATAVIEW [8], which facilitate and automate the complex execution process of a workflow on top of a high-performance computing infrastructure through a user-friendly interface or an API.

In the last few years, cloud computing has become a promising paradigm over the Internet for scientific computing applications. Cloud computing [9] comes with numerous benefits: elasticity, the pay-as-you-go pricing model, virtually unlimited capacity of compute resources, etc., which are particularly suitable for exploratory scientific computing applications. Scientific workflow management systems have started to embrace cloud computing, especially Infrastructure-as-a-Service (IaaS), as their new underlying computing platforms to deliver high-performance computing capabilities to scientific applications. IT resources are often encapsulated as virtual machines (VMs) in cloud computing, which are leveraged by scientific workflow systems to facilitate scientific workflow applications. In order to fulfill different applications' needs, cloud providers are offering various VM types with different computing capacities. For example, the Amazon commercial cloud platform provides 278 VM instance types under five instance families [10]. The complex workflow structure and flexible computing resource selection makes workflow scheduling challenging in the cloud.

Scheduling algorithm plays an essential component in workflow management systems [11] [12]. As a matter fact, workflow scheduling is a well-known NP-complete [13] problem even in the simplest case in which tasks are assigned to an arbitrary number of resources. An ideal scheduling algorithm is expected to generate a scheduling solution of minimal monetary cost and minimal makespan of the workflow's execution, which is an optimization problem of conflicting objectives. How to balance between makespan and execution cost is the main challenge in workflow scheduling. Such a challenge is typically addressed by dividing the workflow scheduling problem into multiple subproblems: deadline-constrained workflow scheduling [14], budget-constrained workflow scheduling [15], and deadline-budget optimization workflow scheduling [16] [17]. This paper focuses on deadline-constrained scheduling of workflows in an IaaS cloud computing environment. Given the workflow execution deadline, δ, our scheduling objective is to find a workflow schedule that results in the minimum execution cost. In order to achieve this objective, our scheduling algorithm needs to address

- *Changxin Bai is a student in the Department of Computer Science, Wayne State University, Detroit, MI. E-mail: changxin@wayne.edu.*
- *Shiyong Lu is with the Department of Computer Science, Wayne State University, Detroit, MI. E-mail: shiyong@wayne.edu.*
- *Dunren Che is with the Department of Computer Science, Southern Illinois University, Carbondale, IL. E-mail: dche@cs.siu.edu.*
- *Junwen Liu is with the Department of Computer Science, Wayne State University, Detroit, MI. E-mail: junwen@wayne.edu.*

the following issues: Given a workflow, how many VM instances are needed for its execution? When should these VM instances be provisioned and de-provisioned? How do we find the workflow schedule, sch, that minimizes the overall monetary cost? When does each of the workflow tasks start and finish? Extended from our earlier work [18], this paper makes a full presentation of our workflow scheduling approach, strategy, algorithms, and experiments. Our scheduling algorithm is path-based – it decides whether two consecutive tasks in a path of a given workflow should be scheduled to the same VM instance or different ones by taking a holistic consideration of the tasks' execution times, data transfer times between the tasks, and the options of the billing cycles of the underlying IaaS cloud platform, while IC-PCP [19] and SGX-E2C2D [20] always assign a whole path to one VM instance and thus are inferior to LPOD on performance. This paper makes the following new contributions in addition to those in [18]:

1) We propose an innovative dynamic programming algorithm with a detailed recurrence relationship explained in algorithm $AssignSuffix$ to make VM assignments to a suffix of a partial critical path with a complexity of $\mathcal{O}(nK^2 + n^2)$, which is much more efficient than its preliminary version [18] with a complexity of $\mathcal{O}(nK^n + n^2)$.
2) We introduce six theorems to prove the correctness of the $AssignSuffix$ algorithm, particularly in terms of the optimization properties and the correctness of the recurrence relationship.
3) Additional experiments have been conducted using a provenance collection module for DATAVIEW to collect runtime provenance information of workflow execution, which is used to better estimate inputs and parameters for LPOD. As a result, the performance and robustness of LPOD are further improved.

This paper is structured as follows. Section 2 introduces basic definitions to formalize the deadline-constrained workflow scheduling problem. Section 3 expounds our proposed scheduling algorithm, LPOD, in detail. Our experimental results are presented and discussed in Section 4. Section 5 reviews related work regarding workflow scheduling. Finally, Section 6 concludes this paper.

2 Problem Description

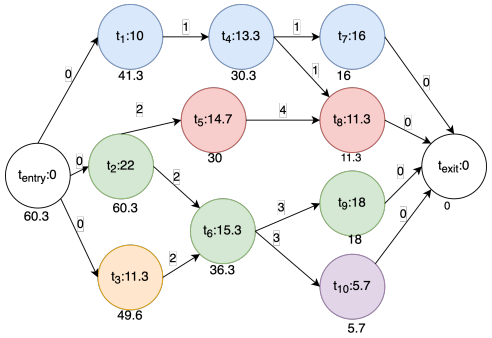


Fig. 1: A sample workflow graph with average execution times on nodes, average data transfer times on edges and priority ranks right below the nodes.

In this section, we introduce a few definitions and formalize the deadline-constrained workflow scheduling problem. The goal of our scheduling is to find workflow schedule sch for given workflow W that minimizes the monetary cost of the workflow's execution in the cloud under given deadline constraint δ. The underlying assumption we make is that only one task is assigned to one VM instance at a time (i.e., no parallelism within one VM instance).

In our modeling, we consider the following aspects of a cloud computing environment and of a given workflow: the capacity of each VM type, the execution time of each workflow task with each VM type, the data transfer rate and price in the cloud, the sizes of data products for a workflow and the booting times of VM instances. We first model a big data workflow and its cloud computing execution environment in the following.

TABLE 1: Execution time matrix.

	t_{entry}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{exit}
VMT^1	0	4	10	6	8	6	8	10	6	10	3	0
VMT^2	0	10	24	10	12	17	16	16	13	16	6	0
VMT^3	0	16	32	18	20	21	22	22	15	28	8	0
$\overline{ET}(t_i)$	0	10	22	11.3	13.3	14.7	15.3	16	11.3	18	5.7	0

To illustrate some modeling notations, we introduce a sample workflow, W, shown in Fig. 1 and assume a general cloud computing environment, C, which provides three VM types. The billing cycle of C is set to 10, and the unit costs for the three VM types are 0.05, 0.02 and 0.01, respectively. The booting time of a VM instance is set to 1. The execution time of each task of workflow W in cloud computing environment C is given in Table 1.

Definition 2.1 (Cloud Computing Environment): A cloud computing environment is modeled as an eight-tuple $C(VMT, VMC, VPrice, VMI, Type, DTR, DPrice, BootDelay)$, where VMT is a set of VM types in the environment, $VMC : VMT \rightarrow \mathbb{R}^+$ is a VM capacity function, $VPrice : VMT \rightarrow \mathbb{R}^+$ is a VM price function, VMI is a set of VM instances, $Type : VMI \rightarrow VMT$ is a VM type function, $DTR : VMI \times VMI \rightarrow \mathbb{R}^+$ is a data transfer rate function, $DPrice : VMI \times VMI \rightarrow \mathbb{R}^+$ is a data transfer price function, and $BootDelay$ is the delayed time after starting a VM instance until it is actually available.

Definition 2.2 (Big Data Workflow): A Big data workflow is modeled as a four-tuple $W(T, D, TSize, Dsize)$, where T is the set of tasks in W, $D \subseteq T \times T$ is a set of data dependency relationship between tasks, $TSize : T \rightarrow \mathbb{R}^+$ is a task size function, and $DSize : D \rightarrow \mathbb{R}^+$ is a data product size function.

To ease presentation, for each workflow, we add two auxiliary tasks, t_{entry} and t_{exit}, and connect them to the original entry tasks (no parents) and exit tasks (no children), respectively, with auxiliary edges (with $DSize(D_{i,j}) = 0$). Next, we introduce the workflow graph definition to model the performance properties of a given workflow W in a particular cloud computing environment C.

Definition 2.3 (Workflow Graph): Given cloud computing environment C and workflow W, a workflow graph is formalized as a five-tuple $G(T, D, ET, DTT, DTC)$, where T is the set of tasks in W, D is the set of data dependencies

among tasks in W, $ET : T \times VMT \rightarrow \mathbb{R}^+$ is a task execution time function, $\overline{ET} : T \rightarrow \mathbb{R}^+$ is the average task execution time function, $DTT : D \times VMI \times VMI \rightarrow \mathbb{R}^+$ is the data transfer time function, $\overline{DTT} : D \rightarrow \mathbb{R}^+$ is the average data transfer time function, and $DTC : D \times VMI \times VMI \rightarrow \mathbb{R}^+$ is the data transfer monetary cost function.

We now model the notion of a workflow schedule, which will be generated by a particular workflow planner in a workflow system, such as the one to be introduced in the next section.

Definition 2.4 (Workflow Schedule): Given cloud computing environment C and workflow W, a workflow schedule is modeled as an eight-tuple $sch(T, VMI, M, AST, AFT, Prov, Deprov, Available)$, where T is the set of tasks in workflow W, VMI is a set of provisioned VM instances in cloud C, $M : T \rightarrow VMI$ is a task-VM mapping function, $AST : T \rightarrow \mathbb{R}^+$ is an actual start time function of a task, $AFT : T \rightarrow \mathbb{R}^+$ is an actual finish time function of a task, $Prov : VMI \rightarrow \mathbb{R}^+$ is a VM instance provisioning time function, $Deprov : VMI \rightarrow \mathbb{R}^+$ is a VM instance deprovisioning time function, and $Available : VMI \rightarrow \mathbb{R}^+$ is a VM instance available time point function.

The total execution time and monetary cost of a schedule sch are defined by workflow makespan and workflow cost, respectively, as follows.

Definition 2.5 (Workflow Makespan *MS*): Given workflow schedule sch, the makespan of workflow schedule sch is defined as:

$$MS(sch) = sch.AFT(sch.t_{exit}) - sch.AST(sch.t_{entry}) \quad (1)$$

Definition 2.6 (Workflow *Cost*): Given workflow schedule sch, the total execution monetary cost of sch in cloud computing environment C is defined as:

$$Cost(sch) = \sum_{v_m \in VMI} VPrice(Type(v_m)) * \lceil \frac{Deprov(v_m) - Prov(v_m)}{l} \rceil \quad (2)$$

The above definition assumes a zero data movement monetary cost among VM instances, which is the data transferred model we are using based on Amazon EC2 that data transferred between Amazon EC2 in the same availability zone is free. The above definition also rounds up VM instance usage to its nearest full billing cycle. Finally, our scheduling problem can be formulated as follows.

Definition 2.7 (Deadline-constrained Workflow Scheduling Problem): Given a workflow W, an IaaS computing environment C, and a deadline δ, the deadline-constrained workflow scheduling problem is to find the optimal workflow schedule sch_{opt} that minimizes the monetary cost of running workflow W within deadline δ:

$$sch_{opt} = \underset{sch}{\operatorname{argmin}} Cost(sch) \quad subject\ to\ Makespan(sch) \leqslant \delta \quad (3)$$

3 The Proposed Workflow Scheduling Algorithm

We have set up an adequate context through the prior sections for the design and detailed presentation of our proposed workflow scheduling algorithm, LPOD. Our algorithm accordingly consists of two phases: the partial critical path construction phase and the resource selection phase. In the first phase, a workflow graph G is partitioned into a set of partial critical paths; in the second phase, a cost-minimizing resource selection scheme is pursued for each path.

In the following, we first introduce some fundamental definitions to facilitate the presentation of the proposed algorithm. We then present and illustrate the algorithm using a representative example, followed by a complexity analysis of its execution time.

3.1 Basic Definitions

The priority rank of a task is introduced to measure the length of the partial critical path from a task to the exit node t_{exit} , which is defined as follows:

$$Pri(t_i) = \begin{cases} 0, & \text{if} t_i = t_{exit}; \\ \overline{ET}(t_i) + \max\limits_{t_j \in (t_i)^\bullet} (\overline{DTT}(D_{i,j}) + Pri(t_j)), & \text{otherwise.} \end{cases} \quad (4)$$

where $(t_i)^\bullet$ denotes the collection of all the child tasks of t_i in G.

The minimum execution time of a task is defined as the execution time of the task on an instance of the fastest VM type in C, which is defined as

$$MET(t_i) = \min_{k \in [1 \ldots K]} ET(t_i, VMT^k) \quad (5)$$

Finally, the estimated execution time of a task t_i is defined as:

$$ET^*(t_i) = \begin{cases} MET(t_i), & \text{if } M(t_i) = null; \\ ET(t_i, Type(M(t_i))), & otherwise. \end{cases} \quad (6)$$

3.2 An overview of $LPOD$

LPOD starts with the main algorithm, $Workflow Planner\_LPOD$, which identifies all the partial critical paths of a workflow by algorithm $FindAllPCPs$ and then makes VM assignments for each path by algorithm $AssingPCPs$. Algorithm $AssignPCPs$ iterates over each partial critical path P_i, assigning as many tasks as possible in the prefix of P_i to existing VM instances first (via algorithm $AssignPrefix$), then to new VM instances (via algorithm $AssignSuffix$). Algorithm $AssignSuffix$ applies a dynamic programming strategy to find the optimal VM assignments for the suffix of P_i: it first uses algorithm $BuildTB$ to store partial solutions, calculated by algorithm $CreateNewTuple$, in a bottom-up fashion, and then uses algorithm $ComputeOptimum$ to identify an optimal solution for the suffix which in turn calls algorithm $AssignSchedule$ to update the workflow schedule sch. Each algorithm is explained in detail in the subsequent subsections.

3.3 The $Workflow Planner\_LPOD$ Algorithm

Algorithm $Workflow Planner\_LPOD$ is the main algorithm of LPOD. As shown in Algorithm 1, the algorithm, after some initialization (lines $1-3$), first computes LFT (Latest Finish Time) of all tasks except t_{exit} in G per Equation 7:

$$LFT(t_i) \leftarrow \min_{t_c \in (t_i)^\bullet} (LFT(t_c) - ET^*(t_c) - \overline{DTT}(D_{i,c})) \quad (7)$$

Algorithm 1: WorkflowPlanner_LPOD

Input : Workflow graph $G(T, E)$ and deadline δ
Output: Workflow schedule sch

1 $sch \leftarrow \emptyset$;
2 $EST(t_{entry}) \leftarrow 0$; $EFT(t_{entry}) \leftarrow 0$;
3 $LFT(t_{exit}) \leftarrow \delta$;
4 Compute LFT for all tasks ;
5 **if** $LFT(t_{entry} < 0)$ **then**
6 | Return $NULL$;
7 **end**
8 Compute EST and EFT for all tasks ;
9 Compute the priority ranks of all tasks;
10 $tlist \leftarrow$ all tasks in G sorted in priority rank decreasing order except for t_{entry} and t_{exit} ;
11 $PP \leftarrow FindAllPCPs(tlist)$;
12 Mark t_{entry} and t_{exit} as assigned;
13 $AssignPCPs(PP, sch)$;

Algorithm 2: FindAllPCPs

Input : Sorted task list $tlist = [t_1, \cdots, t_n]$
Output: Path list PP

1 $PP \leftarrow \emptyset$;
2 **foreach** $t_i \in tlist$ **do**
3 | $P \leftarrow \emptyset$;
4 | $P.add(t_i)$;
5 | $tlist.remove(t_i)$;
6 | **while** t_i *has children in* $tlist$ **do**
7 | | $t_i \leftarrow t_i's$ first child in $tlist$;
8 | | $P.add(t_i)$;
9 | | $tlist.remove(t_i)$;
10 | **end**
11 | $PP.add(P)$;
12 **end**
13 Return PP ;

Once $LFT(t_{entry})$ is computed, the algorithm can immediately decides whether it is possible to generate a feasible schedule for the input workflow G with the given deadline δ on the adopted cloud computing platform. If $LFT(t_{entry}) < 0$, a feasible schedule for this workflow does not exist, which normally means the deadline given by the user is too close and unsatisfiable, and the user should be prompted for a new, more relaxing deadline. Our pseudocode shown in Algorithm 1 in this case simply returns $NULL$ (or an empty schedule, lines $5-7$). Line 8 computes EST (Earliest Start Time) and EFT (Earliest Finish Time) of all tasks. ESTs of all tasks except t_{entry} are computed by Eq. 8 as follows:

$$EST(t_i) \leftarrow \max_{t_p \in {}^\bullet(t_i)} (EST(t_p) + ET^*(t_p) + \overline{DTT}(D_{p,i})) \quad (8)$$

where ${}^\bullet(t_i)$ denotes the parent tasks of t_i in G. EFT is computed by Eq. 9 as follows:

$$EFT(t_i) \leftarrow EST(t_i) + ET^*(t_i) \quad (9)$$

After that, at line 9, the algorithm computes the priority rank per Eq. 4 for each task and sorts all tasks into a list, $tlist$, in decreasing order of their priority ranks (excepting tasks t_{entry} and t_{exit}). Then, the algorithm calls the $FindAllPCPs$ algorithm (line 11) to construct all the partial critical paths and store them in list PP. Lastly, the algorithm marks the two special tasks, t_{entry} and t_{exit}, as $assigned$, and calls $AssignPCPs$ to assign the tasks in each partial critical path to the best fit VM instances, resulting in the final optimal schedule sch that is guaranteed to render the minimum monetary cost for the workflow G and meet its deadline δ.

Applying Algorithm $Workflow Planner\_LPOD$ to the sample workflow introduced in Sec. 2, the computed priority rank of each task is shown right underneath each node in Fig. 1 for for δ = 50.

3.4 The $FindAllPCPs$ algorithm

Before delving in the detail of the $FindAllPCPs$ algorithm, a formal definition of the partial critical path is given below:

Definition 3.1 (Partial Critical Path PCP): Given a workflow graph G and a task t_i in G, t_c is the critical child of unassigned task t_i (t_i has not been assigned to a VM instance), named $CriticalChild(t_i)$ iff: 1) t_c is an unassigned child of t_i; and 2) among all the unassigned children of t_i, t_c has the highest priority value, $Pri(t_c)$. The partial critical path of t_i is defined as $PCP(t_i) = t_i + PCP(CriticalChild(t_i))$ if t_i has a critical child, otherwise, $PCP(t_i) = t_i$, where "+" denotes the concatenation of two paths.

Algorithm $FindAllPCPs$ constructs all partial critical paths based on the priority rank of each task computed per Eq. (4). As shown in Algorithm 2, the algorithm takes a sorted list of workflow tasks as input and returns a list of all paths in the workflow. The output path list PP is initialized as empty at line 1. Then the for loop (starting at line 2) iterates over the input task list $tlist$, builds each local path by consuming and eventually emptying the input task list. Within the for loop, the path variable P is initially set empty for each potential path (line 3), then the current task t_i is added to partial critical path P under construction (line 4) and is removed from task list $tlist$ (line 5); after that, the while loop (lines 6-10) grows each partial critical path by recursively chasing the parent-child relation embedded in the input workflow graph and removes each visited task from the task list $tlist$. The completed partial critical path, P, is then added to the output path list PP (line 11). And finally, at line 13 the algorithm returns the list of all partial critical paths, PP.

3.5 The $AssignPCPs$ algorithm

Algorithm $AssignPCPs$ sequentially schedules all partial critical paths by assigning a prefix of a partial critical path firstly and then the suffix of the path. The $AssignPCPs$ algorithm takes all the partial critical paths identified by algorithm $FindAllPCPs$ as input and returns the workflow schedule sch. For each path $P_i = [t_1, \ldots, t_n]$, we use two auxiliary algorithms to schedule the tasks of P_i to a set of VM instances. Algorithm $AssignPrefix$ (to be elaborated in the next subsection) is called to assign a prefix of P_i, $[t_1, \ldots, t_{li}]$, to existing VM instances without additional computing cost by leveraging the residual idle time of existing VM instances (line 2). Unassigned suffix

Algorithm 3: AssignPCPs

Input : Path list $PP = [P_1, \cdots, P_m]$
InOut: Workflow schedule sch

```
1  foreach P_i = [t_1, ..., t_n] ∈ PP do
2      P_suffix ← AssignPrefix(P_i, sch);
3      if P_suffix ≠ NULL then
4          AssignSuffix(P_suffix, sch);
5      end
6      foreach t_j ∈ P_i do
7          EST(t_j) ← sch.AST(t_j);
8          LFT(t_j) ← sch.AFT(t_j);
9          EFT(t_j) ← sch.AFT(t_j);
10         update ESTs and EFTs of all successors of
             t_j by Eq. (8) and Eq. (9);
11         update LFTs of all predecessors of t_j by
             Eq. (7);
12     end
13 end
```

of P_i, $[t_{li+1}, \ldots, t_n]$, will be returned by the $AssignPrefix$ algorithm, and if it is not empty, algorithm $AssignSuffix$ (to be elaborated in the next section) will be called to schedule the suffix path (lines $3-5$). Once all tasks in a path P_i are scheduled sequentially, the actual start time and actual finish time of the tasks are determined. For any task t_j on path P_i, its EST is then updated to its actual start time and its the EFT and LFT are both updated to its actual finish time (lines $7-9$). The algorithm then updates the EST and EFT of each unassigned successor task t_c of t_j (line 10) using the two equations, Eq. (8) and Eq. (9). After that, the LFT of each unassigned predecessor task t_p of t_j (line 10) is updated using Eq. (7).

3.6 The $AssignPrefix$ algorithm

Algorithm $AssignPrefix$ tries to assign as many tasks as possible in the prefix of a partial critical path to one applicable existing VM instance. The $AssignPrefix$ algorithm takes a partial critical path, P, and a workflow schedule, sch, as the inputs and returns a partial workflow scheduling sch and a suffix path. Two conditions will be checked to decide whether task t_i can be assigned to an existing VM instance, v, with sufficient residual time available based on the task's temporal actual start time:

Condition 1: $ast + ET(t_i, Type(v)) \leq LFT(t_i)$. A task must be finished before its latest finish time on the assigned VM instance.
Condition 2: $ast + ET(t_i, Type(v)) \leq Deprov(v)$. A task must be finished before the deprovisioning time of v.

Algorithm $AssignPrefix$ starts with the first task in P_i, and tries every provisioned VM instance in $sch.VMI$ for possible instance. For each task t_i in P, a temporal actual start time, denoted by ast, is set to the greater of t_i's EST and the time point when v becomes available (lines $5-9$). Once a task can be assigned to an applicable VM instance satisfying the above two conditions (line 10), attributes such as the task set, the actual start time function, the actual finish time function, the task assignment function and the VM instance available time function in workflow schedule sch are updated at lines $11-15$. The length of the processed prefix, li, is also updated at line 16. Then, the execution flow leaves the inner loop (line 18) and continues with the outer loop to try the next task in the path. This process stops until encountering the first task in P_i that cannot be assigned to an applicable VM instance and the execution then leaves the outer loop (lines $21-23$). Lastly, when the value of li is smaller than n, i.e. there exists a non-empty unassigned suffix, P_i, the suffix is returned (lines $25-28$). Otherwise, the algorithm returns $NULL$ at the end.

Algorithm 4: AssignPrefix

Input : Path $P = [t_1, \cdots, t_n]$ and workflow schedule sch
Output: Workflow schedule sch and a suffix of P, P_{suffix}

```
1  li ← 0;
2  foreach t_i ∈ [t_1, ..., t_n] do
3      found ← false ;
4      foreach v_i ∈ sch.VMI do
5          if EST(t_i) ≤ sch.Available(v_i) then
6              ast ← sch.Available(v_i);
7          else
8              ast ← EST(t_i);
9          end
10         if ast + ET(t_i, Type(v_i)) ≤ LFT(t_i) and
             ast + ET(t_i, Type(v_i)) ≤ Deprov(v_i) then
11             sch.T.insert(t_i);
12             sch.AST.insert(t_i, ast);
13             sch.AFT.insert(t_i, ast +
                 ET(t_i, Type(v_i)));
14             sch.M.insert(t_i, v_i);
15             sch.Available.update(v, ast +
                 ET(t_i, Type(v_i));
16             li ← i;
17             found ← true;
18             break;
19         end
20     end
21     if found == false then
22         break;
23     end
24 end
25 if li < n then
26     P_suffix ← [t_li+1, ..., t_n] ;
27     Return P_suffix ;
28 end
29 Return NULL;
```

3.7 The $AssignSuffix$ algorithm

Algorithm 5: AssignSuffix

Input : Path $P = [t_1, \cdots, t_n]$
InOut: Workflow schedule sch

```
1  TB ← BuildTB(P);
2  ComputeOptimum(TB, sch);
```

Algorithm $AssignSuffix$ uses a dynamic programming strategy, which tries to minimize the accumulated billing cost for assigning one path to a set of possible VM instances (or only one instance, which is a special case) to find the optimal assignment for a suffix path. It takes a suffix path P returned by the $AssignPrefix$ algorithm and workflow schedule sch as inputs and returns an updated (optimal) workflow schedule sch. The optimal assignment of a path can be simplified by finding the optimal assignments of the path with the last task t_i assigned to a best-suitable VM instance of type VMT^k. Obviously, the VM assignment of task $t_i (i \geq 2)$ must consider and be based on a direct parent task t_{i-1}'s assignment. The optimal solution of assigning t_i to a VM instance, v, of type VMT^k, can only be obtained through one of the following two cases:

Case 1: Instance v, of type VMT^k, is an existing instance assigned to t_{i-1}, so t_i is assigned to the same instance v.

Case 2: Instance v, of type VMT^k, is a new instance launched for t_i, while t_{i-1} (if exists) is assigned to another instance u of a different type other than VMT^k.

Definition 3.2 $\boldsymbol{Sol\_o1(t_i,k)}$ **&** $\boldsymbol{Sol\_o2(t_i,k)}$ For each $k \in [1 \cdots K]$, we define $Sol\_o1(t_1,k) = Sol\_o2(t_1,k)$, which is the solution that assigns t_1 to a new VM instance of type VMT^k. For each $i \in [2 \cdots n]$ and $k \in [1 \cdots K]$, we define $\boldsymbol{Sol\_o1(t_i,k)}$ as the best candidate solution for t_i that satisfies two conditions: i) assigns t_i and t_{i-1} to the same instance of type VMT^k and ii) has the smallest accumulated cost for the partial path, $[t_1, \cdots, t_i]$. For each $i \in [2 \cdots n]$ and $k \in [1 \cdots K]$, we define $\boldsymbol{Sol\_o2(t_i,k)}$ as the best candidate solution for t_i that satisfies two conditions: i) assigns t_i to a new instance of type VMT^k, which is different from the type, VMT^j $(j \neq k)$, of the assigned instance of t_{i-1} and ii) has the smallest accumulated cost for the partial path, $[t_1, \cdots t_i]$.

Definition 3.3 $\boldsymbol{Sol\_o(t_i,k)}$**:** For each $i \in [1 \cdots, n]$ and each $k \in [1, \ldots, K]$, we define $Sol\_o(t_i,k)$ as the best solution for t_i that has the smallest accumulated cost for the partial path, $[t_1 \cdots t_i]$.

For each $i \in [2 \cdots n]$, $Sol\_o1(t_i,k)$ and $Sol\_o2(t_i,k)$ for task t_i are built on $Sol\_o1(t_{i-1},j)$ and $Sol\_o2(t_{i-1},j)$ $(j \in [1 \cdots K])$ of task t_{i-1}. There are 2 candidate solutions to compute $Sol\_o1(t_i,k)$ and $2K-2$ candidate solutions to compute $Sol\_o2(t_i,k)$ Here K is the number of VM types offered by the underlying cloud platform.

In order to calculate the optimal solution of $t_1 \ldots t_n$ with t_n assigned to an instance of type VMT^k, a bottom-up approach starts from t_1 with K possible assignments and uses those $2K$ solutions of t_i to build solutions for t_{i+1} until i equals to $n-1$. This process is visualized in Fig. 3. However, we need to maintain the two best candidate solutions of each t_i assigned to an instance of type VMT^k because the best solution of t_i assigned to an instance of type VMT^k is not necessarily computed based on the best solution of t_{i-1} assigned to an instance of VMT^k, but necessarily computed based on the total $2K$ solutions of t_{i-1}, two for each VM type VMT^k, which can be proved by the following three theorems.

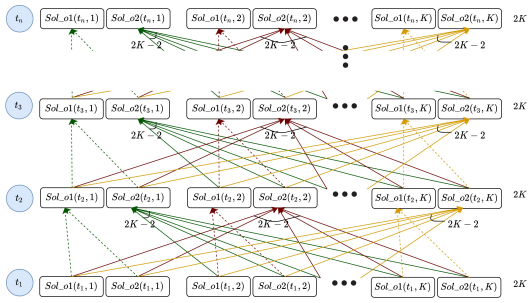


Fig. 3: The recurrence relationship for a path $P = [t_1, \cdots, t_n]$. K is a constant that represents the number of VM types.

Theorem 3.1. *For each $k \in [1 \ldots K]$, the best solution of t_2 for VMT^k is necessarily computable from the best solution of t_1 for VMT^k.*

Proof. For each $k \in [1 \ldots K]$, we have $Sol\_o(t_1,k) = Sol\_o1(t_1,k) = Sol\_o2(t_1,k)$. Since t_2's assignment is based on $Sol\_o1(t_1,k)$ and $Sol\_o2(t_1,k)$, the best solution of t_2 for VMT^k is necessarily computable from the best solution of t_1 for VMT^k. □

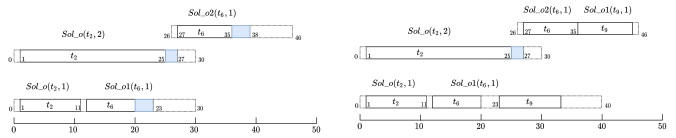
Theorem 3.2. *For each $k \in [1 \ldots K]$, and each $i \in [2 \cdots n-1]$, the best solution of t_{i+1} for VMT^k is not necessarily computable from the best solutions of t_i for VMT^j $(j \in [1 \ldots K])$.*

Proof. We prove by example (in addition to the general truth that two local optimal solutions do not necessarily make a global optimal solution). For the sample partial critical path $t_2 \rightarrow t_6 \rightarrow t_9$, Fig. 4 shows the assignment of the tasks t_2 and t_6 in turn. The upper left chart shows the solution of t_6, $Sol\_o2(t_6,1)$, based on the best solution of t_2, $Sol\_o(t_2,2)$; the lower left chart shows shows the solution of t_6, $Sol\_o1(t_6,1)$, based on the best solution of t_2, $Sol\_o(t_2,2)$. The cost of $Sol\_o1(t_6,1)$ (using an instance of VMT^1 executing t_2 and t_6 with three billing cycles) is 0.15 and the cost of $Sol\_o2(t_6,1)$ (using an instance of VMT^2 executing t_2 with three billing cycles and an instance of VMT^1 running t_6 with two billing cycles) is 0.16, so $Sol\_o1(t_6,1)$ is the best solution of t_6. The upper right chart shows the solution of t_9, $Sol\_o1(t_6,1)$, based on the solution of t_6, $Sol\_o2(t_6,1)$, whose cost is still 0.16, which is smaller than assigning t_9 based on the best solution of t_6, $Sol\_o1(t_6,1)$ with one billing cycle added. So, it suffices to say generally, the best solution of t_{i+1} for VMT^k is not necessarily computable from the best solution of t_i for VMT^j $(j \in [1 \ldots K])$. □

Theorem 3.3. *For each $k \in [1 \ldots K]$, and each $i \in [1 \cdots n-1]$, each $Sol\_o1(t_{i+1},k)$ is necessarily computable from $Sol\_o1(t_i,k)$ and $Sol\_o2(t_i,k)$ and $Sol\_o2(t_{i+1},k)$ are necessarily computable from $Sol\_o1(t_i,j)$ and $Sol\_o2(t_i,j)$ $(j \in [1 \ldots K]$ & $j \neq k)$.*

Proof. The proof is by induction on i.

Base case $i = 1$**:** For each $k \in [1 \ldots K]$, we have $Sol\_o(t_1,k) = Sol\_o1(t_1,k) = Sol\_o2(t_1,k)$. Since t_2 can use some residual time of the instance executing t_1 and there are two best candidate solutions of t_1 for VMT^k,



(a) Two best assignments of t_6 on a VM instance of type VMT^1.

(b) The optimal assignment of t_9 of type VMT^1 and assigning t_9 based on $Sol\_o1(t_6, 1)$.

Fig. 4: The gantt charts represent task assignments on each VM instance with filled box showing the output data transfer time and dashed boxes showing the idle time on an instance.

i.e., $Sol\_o1(t_1, k)$ and $Sol\_o2(t_1, k)$, $Sol\_o1(t_2, k)$ is the best candidate solution by assigning t_2 to one of the solutions $Sol\_o1(t_1, k)$ and $Sol\_o2(t_1, k)$ with a smaller accumulated cost. In this case, $Sol\_o1(t_2, k)$ is necessarily computable from $Sol\_o1(t_1, k)$ and $Sol\_o2(t_2, k)$. There are $2K - 2$ best candidate solutions of assigning t_2 in a different instance from t_1 assigned an instance of VMT^k. $Sol\_o2(t_2, k)$ is based on the assignment that t_1 is assigned to a different instance of VMT^k having the smallest accumulated cost, which is one among $Sol\_o1(t_1, j)$ and $Sol\_o2(t_1, j)(j \in [1 \dots K]\ \&\ j \neq k)$. The statement is true for $i = 1$.

Induction step: Suppose $i = l$ is true that $Sol\_o1(t_{l+1}, k)$ is necessarily computable from $Sol\_o1(t_l, k)$ and $Sol\_o2(t_l, k)$, and $Sol\_o2(t_{l+1}, k)$ is necessarily computable from $Sol\_o1(t_l, j)$ and $Sol\_o2(t_l, j)$ $(j \in [1 \dots K]\ \&\ j \neq k)$. So there are $2K$ best candidate solutions of t_{l+1}, two best candidate solutions for each $k \in [1 \dots K]$. When $t_{(l+1)+1}$ is assigned to the same instance as t_{l+1}, which is the instance in $Sol\_o1(t_{l+1}, k)$ or $Sol\_o2(t_{l+1}, k)$, $Sol\_o1(t_{(l+1)+1}, k)$ is calculated based on $Sol\_o1(t_{l+1}, k)$ and $Sol\_o2(t_{l+1}, k)$. When $t_{(l+1)+1}$ is assigned to a different instance from t_{l+1}, which is the instance in $Sol\_o1(t_{l+1}, j)$ or $Sol\_o2(t_{l+1}, j)(j \in [1 \dots K]\ \&\ j \neq k)$, $Sol\_o2(t_{(l+1)+1}, k)$ is calculated based on $Sol\_o1(t_{l+1}$ or $Sol\_o2(t_{l+1}, j)(j \in [1 \dots K]\ \&\ j \neq k)$, $Sol\_o2(t_{(l+1)+1}, k)$. So, the statement is true for $i = l + 1$.

Conclusion: By the principle of induction, for $i = 1 \dots n - 1$, $Sol\_o1(t_{i+1}, k)$ is necessarily computable from $Sol\_o1(t_i, k)$ and $Sol\_o2(t_i, k)$, and $Sol\_o2(t_{i+1}, k)$ is necessarily computable from $Sol\_o1(t_i, j)$ and $Sol\_o2(t_i, j)(j \in [1 \dots K]\ \&\ j \neq k)$.

□

The optimal assignment of path $P = [t_1, \cdots, t_n]$ with t_n assigned to an instance of VMT^k is dependent on the $2K$ solutions of t_{n-1}. Specifically, this strategy is implemented through a tabular form by generating the best candidate solutions of t_i on the solutions of t_{i-1}, represented as table TB. The algorithm uses a two-stage procedure to find the optimal assignment. The first stage is to build TB using the $BuildTB$ algorithm, and the second stage is to traverse back TB to find the best assignments using the $ComputeOptimum$ algorithm. The two auxiliary algorithms, $BuildTB$ and $ComputeOptimum$, are elucidated in the following subsections.

3.8 The $BuildTB$ algorithm

Algorithm 6: BuildTB

Input : Path $P = [t_1, \cdots, t_n]$
Output: Table TB

1 $TB \leftarrow \emptyset$;
2 **foreach** $VMT^k \in VMT$ **do**
3 $nt \leftarrow CreateNewTuple(1, k, NULL)$; ▷ First task of P
4 **if** $nt.eft \leq LFT(t_1)$ **then**
5 $TB.insert(nt)$
6 **end**
7 **end**
8 **foreach** $t_i \in [t_2, \cdots, t_n]$ **do**
9 **foreach** $VMT^k \in VMT$ **do**
10 $nt1 \leftarrow NULL$; $cost1 \leftarrow +\infty$;
11 $nt2 \leftarrow NULL$; $cost2 \leftarrow +\infty$;
12 **foreach** *tuple pt of* $t_{i-1} \in TB$ **do**
13 $nt \leftarrow CreateNewTuple(i, k, pt)$;
14 **if** $nt.eft \leq LFT(t_i)$ **then**
15 **if** $nt.VMtype == pt.VMtype$ **then**
16 **if** $nt.cost < cost1$ **then**
17 $cost1 \leftarrow nt.cost$;
18 $nt1 \leftarrow nt$;
19 **end**
20 **else**
21 **if** $nt.cost < cost2$ **then**
22 $cost2 \leftarrow nt.cost$;
23 $nt2 \leftarrow nt$;
24 **end**
25 **end**
26 **end**
27 **end**
28 **if** $nt1 \neq NULL$ **then**
29 $TB.insert(nt1)$
30 **end**
31 **if** $nt2 \neq NULL$ **then**
32 $TB.insert(nt2)$
33 **end**
34 **end**
35 **end**
36 Return TB ;

Algorithm $BuildTB$ constructs table TB, in which each tuple represents a potential assignment of a task. The $BuildTB$ algorithm takes the suffix of a partial critical path identified by the $AssignPrefix$ algorithm as input and returns task assignments table TB, which is based on the recurrence relationship in which a solution of partial critical path $[t_1, \cdots, t_i]$ is built on a solution of its prefix $[t_1, \cdots, t_{i-1}]$. While t_1 is always assigned to a new VM instance, the main point is to decide whether a new VM instance needs to be assigned to t_i. This depends on the potential assignments of $[t_1, \cdots, t_{i-1}]$. For the first task, t_1, the $CreateNewTuple$ algorithm is called to create a new tuple for each VM type VMT^k. Once its temporal actual finish time is not greater than its LFT, the created tuple is inserted into TB as a candidate assignment (lines $2 - 7$). TB maintains at most K solutions for t_1, one for each VM type, VMT^k. Except for the tuples belonging to t_1, at

most $2K$ solutions are maintained in TB for t_i with two best solutions for each VM type VMT^k corresponding to $Sol\_o1(t_i, k)$ and $Sol\_o2(t_i, k)$, represented by $nt1$ and $nt2$, respectively. Solution 1: $nt1$, the best solution that assigns t_i to a VM instance of type VMT^k such that t_{i-1} is also assigned to the same instance as t_i (lines $15-19$). Solution 2: $nt2$, the best solution that assign t_i to a VM instance of type VMT^k while t_{i-1} is assigned to a VM instance different from that of t_{i-1} (lines $20-25$). The process applies to every VM type through the for loop starting at line 9. If these kinds of assignments are found, the two tuples are inserted into TB (lines $28-33$). At the end, TB is returned.

3.9 The $CreateNewTuple$ algorithm

Algorithm 7: CreateNewTuple

Input : Index i of path P, index k of VMT and tuple pt
Output: New tuple nt

1 create new tuple nt;
2 $nt.id \leftarrow genID()$;
3 $nt.t \leftarrow t_i$;
4 $nt.VMtype \leftarrow VMT^k$;
5 $nt.est$ is calculated by Eq. (10) ;
6 $nt.eft$ is calculated by Eq. (11);
7 $nt.maxedge$ is calculated by Eq. (12);
8 $nt.prov$ is calculated by Eq. (13);
9 $nt.deprov$ is calculated by Eq. (14);
10 $nt.cost$ is calculated by Eq. (15) and Eq. (16);
11 Return nt;

Algorithm $CreateNewTuple$ creates a new tuple, denoted by variable nt, capturing the information about a potential assignment of task t_i by extending the potential assignment of a prior task, t_{i-1}, captured in a tuple denoted by variable pt. Algorithm $CreateNewTuple$ takes i, a task index (of partial critical path P), k, a VM type index, and pt, the previous tuple (created for t_{i-1}) as inputs and returns a new tuple, nt. More specifically, TB stores a list of tuples $(id, t, VMtype, est, eft, maxedge, prov, deprov, cost, idref)$ computed in corresponding to task t_i in P. pt represents the tuple created for the parent task t_{i-1} based on which the new tuple nt of t_i is to be computed. The attributes of nt in table TB are:

- id is an unique identifier of each tuple, which is automatically generated by function genID();
- t stores the identifier of task t_i in path P.
- $VMtype$ stores the VM type that is temporarily assigned to t_i in the algorithm.
- est stores the temporary earliest start time of task t_i. It is calculated by equation 10 based on the following three cases:
  i) t_i is the first task: $i = 1$;
  ii) t_i will use the same VM instance as t_{i-1}: $i > 1$ and $pt.VMtype = nt.VMtype$;
  iii) t_i will be assigned to a new VM instance: $i > 1$ and $pt.VMtype \neq nt.VMtype$.

$$nt.est = \begin{cases} EST(t_i), & \text{(i)} \\ \max(EST(t_i), pt.eft), & \text{(ii)} \\ \max(EST(t_i), pt.eft + \overline{DTT}(D_{i-1,i})), & \text{(iii)} \end{cases} \quad (10)$$

Case i) takes the EST of task t_i directly as the temporary earliest start time. Case ii) decides the time point when t_i is ready to be executed without considering the data transfer time from parent task $pt.t$. Case iii) decides the time point when t_i is ready to be executed on a different instance with the transfer time of data from parent task $pt.t$ considered.

- eft stores the temporary earliest finish time of task t_i, which is computed by:

$$nt.eft \leftarrow nt.est + ET(t_i, nt.VMtype) \quad (11)$$

The value of eft is calculate based on the temporary earliest start time of task t_i by adding the execution time of the task on an instance of type $nt.VMtype$.

- $maxedge$ stores the largest data transfer time from t_i to its direct children, which is computed by:

$$nt.maxedge \leftarrow \max_{t_c \in (t_i)^\bullet} \overline{DTT}(D_{i,c}) \quad (12)$$

The $maxedge$ is used to calculate the deprovisioned time of a VM instance. More specifically, if a child task is assigned to a different instance, the current VM instance can only be deprovisioned after all the output data has been transferred to the VM instance running the child task.

- $prov$ stores the temporary provisioning time for a VM instance executing task t_i in nt. It is calculated based on the following two cases:
  i) t_i uses the same VM instance as t_{i-1}: $i > 1$ and $pt.VMtype = nt.VMtype$;
  ii) t_i is the first task assigned to a new VM instance: $i = 1$, or $i > 1$ and $pt.VMtype \neq nt.VMtype$.

$$nt.prov = \begin{cases} pt.prov, & \text{(i)} \\ nt.est - BootDelay, & \text{(ii)} \end{cases} \quad (13)$$

Case i) does not change the provisioning time, because task $pt.t$ and task $nt.t$ are assigned to the same instance. Case ii) sets the provisioning time the same as the first task's temporary earliest start time subtracting the provision delay time $BootDelay$.

- $deprov$ stores the temporal deprovisioning time for a VM instance running task t_i. It is calculated by equation 14 based on the following three cases, where $round(x, l) = \lceil \frac{x}{l} \rceil \times l$, l standing for the length of a billing cycle:
  i) t_i uses the same VM instance as t_{i-1} that has sufficient residual time to accommodate the execution of t_i: $i > 1$, $pt.VMtype = nt.VMtype$ and $nt.eft + nt.maxedge \leq pt.deprov$;
  ii) t_i uses the same VM instance as t_{i-1} that does not have sufficient residual time to accommodate the execution of t_i; therefore, the instance needs to be extended with additional billing cycles to accommodate the execution of t_i: $i > 1$, $pt.VMtype = nt.VMtype$ and $nt.eft + nt.maxedge > pt.deprov$;
  iii) t_i is assigned to a new VM instance as its first task: i = 1, or $i > 1$ and $pt.VMtype \neq nt.VMtype$.

$$\begin{cases} pt.deprov, & \text{(i)} \\ pt.deprov+ \\ round(nt.eft + nt.maxedge - pt.deprov, l), & \text{(ii)} \\ nt.prov+ \\ round(nt.eft + nt.maxedge - nt.prov, l), & \text{(iii)} \end{cases} \tag{14}$$

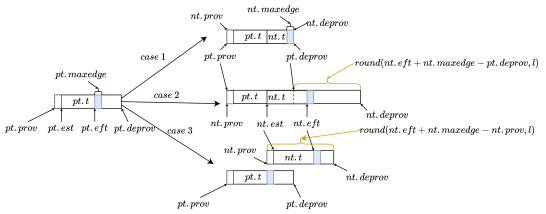


Fig. 5: Assign task $nt.t$ (t_i) based on the assignment of task $pt.t$ (t_{i-1}) in three cases.

Case i) will not change the deprovisioning time of an existing VM instance since the residual time in the existing VM instance is sufficient to accommodate the execution of t_i, which is illustrated in Fig. 5 and marked as case 1. Case ii) essentially adds additional billing cycles (the second term in the equation) on top of the existing deprovisioning time, as illustrated in Fig. 5, marked as case 2. Case iii) adds necessary billing cycles (the second term) to the provisioning time of the new VM instance assigned to t_i, as illustrated in Fig. 5, marked as case 3.

- $cost$ stores the temporal accumulated cost assuming the assignment of $nt.t$ to a corresponding VM instance. It is calculated by equation 15 based on three cases, where $billingcost(x, nt, l) = round(x)/l \times VPrice(nt.VMtype)$ and $adcost(pt, nt, l)$ represents the additional cost of assigning task $nt.t$ based on the assignment captured in the prior tuple pt:

  i) t_i is the first task: $i = 1$;

  ii) t_i is not the first task: $i > 1$.

$$nt.cost = \begin{cases} billingcost(nt.deprov - nt.prov, nt, l), & \text{(i)} \\ pt.cost + adcost(pt, nt, l), & \text{(ii)} \end{cases} \tag{15}$$

Case i) calculates the cost of the new instance based on the provisioned time. Case ii) calculates the cost by adding additional cost of assigning task $nt.t$ baesd on assignment pt.

Then next step is to give the formula how to calculate $adcost(pt, nt, l)$, which is calculated by 16 based on two cases:

i) t_i will use the same VM instance as t_{i-1}: $i > 1$, $pt.VMtype = nt.VMtype$;

ii) t_i will use a new instance different from t_{i-1}: $i > 1$ and $pt.VMtype \neq nt.VMtype$.

$$\begin{cases} billingcost(nt.deprov - pt.deprov, nt, l), & \text{(ii)} \\ billingcost(nt.deprov - nt.prov, nt, l), & \text{(iii)} \end{cases} \tag{16}$$

Case i) essentially calculates the cost for the extended period of the VM instance. Case ii) calculates the cost of the provisioned time of a new instance assigning t_i.

- $idref \leftarrow pt.id$, which stores the id of the tuple of t_{i-1}, the direct parent of task t_i in partial critical path P, based on which the current tuple of t_i is calculated. It is used to trace the chain of the optimal assignments.

Theorem 3.4. *For each $k \in [1 \cdots K]$, and each $i \in [1 \cdots n-1]$, the recurrence relationship between t_{i+1} and t_i is*

$$\begin{aligned} &Sol\_o1(t_{i+1},k).cost= \\ &\min(Sol\_o1(t_i,k).cost+adcost(Sol\_o1(t_i,k),nt,l), \\ &\quad Sol\_o2(t_i,k).cost+adcost(Sol\_o2(t_i,k),nt,l)) \\ &Sol\_o2(t_{i+1},k).cost= \\ &\min_{j\in[1\ldots K]\ \&\ j\neq k}(Sol\_o1(t_i,j).cost+adcost(Sol\_o1(t_i,j),nt,l), \\ &\quad Sol\_o2(t_i,j).cost+adcost(Sol\_o2(t_i,j),nt,l)) \end{aligned} \tag{17}$$

Proof. The proof is by induction on i.

1 **Base case** $i = 1$**:** For each $k \in [1 \cdots K]$, when $i = 1$, we have $Sol\_o(t_1, k) = Sol\_o1(t_1, k) = Sol\_o2(t_1, k)$, so the cost of $So1\_o1(t_1, k)$ and $So1\_o2(t_1, k)$ are calculated based on the VM instance's provisioned time. Since t_2 can use some residual time of the instance executing t_1 and there are two best candidate assignments of t_1 for VMT^k, i.e., $Sol\_o1(t_1, k)$ and $Sol\_o2(t_1, k)$, $Sol\_o1(t_2, k)$ is the solution by assigning t_2 to the one of the two solutions $Sol\_o1(t_1, k)$ and $Sol\_o2(t_1, k)$ with the smaller accumulated cost. In this case, we can have the accumulated cost relationship: $Sol\_o1(t_2, k).cost = \min(Sol\_o1(t_1, k).cost + adcost(Sol\_o1(t_1, k), nt, l), Sol\_o2(t_1, k).cost + adcost(Sol\_o2(t_1, k), nt, l)$. When t_2 and t_1 are assigned to different instances and the accumulated cost of assigning t_2 to a new instance is constant for VMT^k, $Sol\_o2(t_2, k)$ is based on the assignment that t_1 is assigned to a different instance of VMT^k having the smallest accumulated cost, which is the one among $Sol\_o1(t_1, j)$ and $Sol\_o2(t_1, j)(j \in [1 \ldots K]\ \&\ j \neq k)$. In this case, the recurrence relationship between t_2 and t_1 holds.

2 **Induction step:** Suppose for $i = l$ the recurrence relationship between t_{l+1} and t_l is true. The accumulated costs of $Sol\_o1(t_{l+1}, k)$ and $Sol\_o2(t_{l+1}, k)$ are calculated with $k \in [1 \ldots K]$. Then we need to exhibit the accumulated cost relationship between $t_{(l+1)+1}$ and t_{l+1}, considering all potential assignments of t_{l+1} for each $k \in [1 \ldots K]$. By adding the cost of assigning $t_{(l+1)+1}$ to the same instance as t_{l+1}, which is the instance in $Sol\_o1(t_{l+1}, k)$ or $Sol\_o2(t_{l+1}, k)$, so $Sol\_o1(t_{(l+1)+1}, k).cost = \min(Sol\_o1(t_{l+1}, k).cost + adcost(Sol\_o1(t_{l+1}, k), nt, l), Sol\_o2(t_{l+1}, k).cost + adcost(Sol\_o2(t_{l+1}, k), nt, l))$. By adding the cost of assigning $t_{(l+1)+1}$ to the different instance as t_{l+1}, which is the instance in $Sol\_o1(t_{l+1}, j)$ or $Sol\_o2(t_{l+1}, j)$, so $Sol\_o2(t_{(l+1)+1}, k).cost = \min_{j\in[1\ldots K]\ \&\ j\neq k}(Sol\_o1(t_{l+1}, j).cost + adcost(Sol\_o1(t_{l+1}, j), nt, l), Sol\_o2(t_{l+1}, j).cost + adcost(Sol\_o2(t_{l+1}, j), nt, l))$. That is, the recurrence relationship is true for $i = l + 1$.

3 **Conclusion:** By the principle of induction, for $i = 1 \ldots n-1$, the recurrence relationship between t_{i+1} and t_i holds.

□

Theorem 3.5. *For each $k \in [1 \cdots K]$, and each $i \in [1 \cdots n]$, the best solution of t_i for VMT^k, $Sol\_o(t_i, k) = min(Sol\_o1(t_i, k), Sol\_o2(t_i, k))$.*

Proof. We prove by enumeration. For $k \in [1 \cdots K]$, when $i = 1$, we have $Sol\_o(t_1, k) = Sol\_o1(t_1, k) = Sol\_o2(t_1, k)$. It is true that $Sol\_o(t_1, k) = min(Sol\_o1(t_1, k), Sol\_o2(t_1, k))$; when $i \geq 2$, $Sol\_o1(t_i, k)$ represents the best candidate solution for case 1, and $Sol\_o2(t_i, k)$ represents the best cadidate solution for case 2, as there is no other cases, we take whichever of the two cases that yields the minimal accumulated cost at task t_i, i.e., $Sol\_o(t_i, k) = min(Sol\_o1(t_i, k), Sol\_o2(t_i, k))$. Therefore, for all cases ($i = 1 \ldots n$), we have $Sol\_o(t_i, k) = min(Sol\_o1(t_i, k), Sol\_o2(t_i, k))$. □

Theorem 3.6. *The best solution for a path* $P = [t_1 \ldots t_n]$, $Sol\_o(P) = \min\limits_{k \in [1 \ldots K]} (Sol\_o(t_n, k))$.

Proof. We prove by enumeration. Based on Theorem 3.5, $Sol\_o(t_n, k) = min(Sol\_o1(t_n, k),$
$Sol\_o2(t_n, k))$. Since $k \in [1 \ldots K]$, the best solution for a path is to select the smallest accumulated cost by assigning t_n to each instance belonging to each VM type VMT^k. Hence, we have the following best solution for path P $(t_1 \ldots t_n)$: $Sol\_o(P) = \min\limits_{k \in [1 \ldots K]} (Sol\_o(t_n, k))$. □

The correctness of our algorithms $BuildTB$ and $ComputeOptimum$ are backed by the above three theorems.

3.10 The $ComputeOptimum$ algorithm

Algorithm $ComputeOptimum$ finds the tuple of minimal accumulated cost in TB and traces back to all the assignments of the prior tasks in the suffix – together they make up the optimal schedule for the tasks in the suffix. $ComputeOptimum$ takes table TB and (partial) workflow schedule sch as inputs and results in a completed workflow schedule sch. Variable $pret$ (line 1) is the tuple containing the assignment of t_n in path P with the minimal accumulated cost. Workflow schedule sch is updated based on $pret$ through the $AssignSchedule$ algorithm invoked at line 2. During each iteration of the for loop, $pret.t$'s parent tuple is retrieved by the reference index of of $pret$ (line 4), the corresponding optimal assignment is incorporated into the schedule sch (line 5), and the "pointer" $pret$ is advanced to the parent task's tuple in TB (line 6) in order to start the next iteration.

Algorithm 8: ComputeOptimum

Input : Table TB
InOut: Workflow schedule sch

1 $pret \leftarrow$ assign the tuple in TB with $t = t_n$ and minimal cost $cost$;
2 $AssignSchedule(sch, pret, NULL)$;
3 **foreach** $t_i \in [t_{n-1}, \cdots t_1]$ **do**
4 $\quad ct \leftarrow$ select $*$ from TB where $id = pret.idref$;
5 $\quad AssignSchedule(sch, ct, pret)$;
6 $\quad pret \leftarrow ct$;
7 **end**

3.11 The $AssignSchedule$ algorithm

lgorithm $AssignSchedule$ focuses on updating the attributes of the partial workflow schedule sch. It takes two tuples, current tuple ct and previous tuple $pret$, and workflow schedule sch as inputs and returns workflow schedule sch. The task in ct is added to the scheduled task set $sch.T$, followed by updating both its actual start time and actual finish time in sch (lines $1-3$). If the task has no predecessor or the two tasks are assigned to different VM instances (line 4), a new instance is created per the VM type specified in ct (line 5). Line 6 updates the task-to-VM mapping function, followed by updating of the VM pool at line 7. The instance's provisioning time, deprovisioning time, and available time are respectively updated as well based on the attributes in ct (lines $7-10$). If the two contiguous tasks are assigned to the same VM instance (line 12), only the current task's assignment needs be updated to the same instance as that of $pret.t$ (line 13).

Algorithm 9: AssignSchedule

Input : Tuple ct and tuple $pret$
InOut: Workflow schedule sch

1 $sch.T.insert(ct.t)$;
2 $sch.AST.insert(ct.t, ct.est)$;
3 $sch.AFT.insert(ct.t, ct.eft)$;
4 **if** $pret == NULL \,||\, ct.VMtype \neq pret.VMtype$ **then**
5 $\quad v \leftarrow$ assign a new instance from $ct.VMtype$;
6 $\quad sch.M.insert(ct.t, v)$;
7 $\quad sch.VMI.insert(v)$;
8 $\quad sch.Available.insert(v, ct.eft)$
9 $\quad sch.Prov.insert(v, ct.prov)$;
10 $\quad sch.Deprov.insert(v, ct.deprov)$;
11 **end**
12 **if** $ct.VMtype == pret.VMtype$ **then**
13 $\quad sch.M.insert(ct.t, M(pret.t))$;
14 **end**

3.12 Time complexity

Given a workflow graph G with $|T| = n$ and $|D| = e$, we have $e \leq n(n-1)/2$. The first part of the algorithm is to initialize the EST, EFT and LFT of each task in G, which requires a forward and backward processing of all nodes and edges. So its time complexity equals $\mathcal{O}(n + e)$, that is $\mathcal{O}(n^2)$. The time complexity of calculating the priority rank of each task is based on t_{exit}, which needs to traverse G, which equals $\mathcal{O}(n^2)$. Then the procedure $FindAllPCPs$ is called to return all partial critical paths with time complexity equaling $\mathcal{O}(n^2)$. Suppose there are m partial critical paths returned from algorithm $FindAllPCPs$, each path contains n/m tasks. The $AssignPrefix$ algorithm tries all existing instances to schedule a prefix of the path without additional cost. The maximum number of computing resource is equal to the number of tasks, so the time complexity of algorithm $AssignPrefix$ is $\mathcal{O}(n^2/m)$. The time complexity of algorithm $BuildTB$ is $\mathcal{O}(K^2 n/m)$ where K is the number of VM types. The time complexity of algorithm $ComputeOptimum$ is $\mathcal{O}(n)$. Consequently, the overall time

complexity of algorithm $AssignPCPs$ is $\mathcal{O}(n^2 + K^2 n)$, which is also the time complexity of the LPOD algorithm.

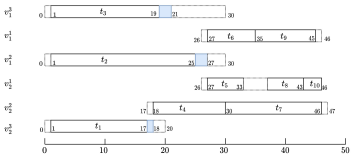


Fig. 6: The scheduling result of sample workflow W by the LPOD algorithm.

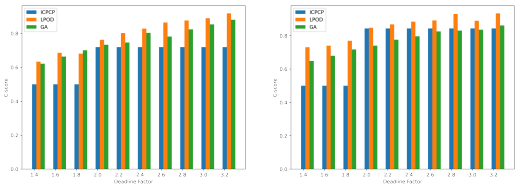
The scheduling result of the whole sample workflow W generated by LPOD is shown in Fig. 6 in the form of Gantt charts. Each provisioned VM instance is listed on the leftmost side. The dashed box shows the provisioned time period of each VM instance with its provisioning and the deprovisioning time points marked; inside the box of each VM instance, a unfilled solid box shows a task's execution period marked with its actual start time and actual finish time, and the filled portion in a solid box indicates transfer time of the output data of the task. Two instances v_1^3 and v_2^3 of VMT^3 are provisioned to execute t_3 and t_1 in a parallel manner; v_1^2 of VMT^2 is provisioned to execute t_2; v_2^2 is provisioned to execute t_4 and t_7 sequentially; v_1^1 and v_2^1 of VMT^1 are scheduled with the same provisioning and deprovisioning time points, with t_6 and t_9 scheduled on v_1^1, and t_5, t_8, and t_{10} on v_2^1. The overall cost of the produced schedule for the sample workflow is 0.37.

4 Performance Evaluation

The proposed workflow scheduling algorithm, LPOD, has been implemented and integrated into the Workflow Engine of DATAVIEW. The algorithm has been thoroughly tested and evaluated with several synthetic workflows and one real workflow. Montage [21] and LIGO [22] are two popular workflows in the workflow scheduling community. As there is no available java code of above two workflows to researchers, we fully implemented the structures of the two workflows, but the computation of each task is simulated by merge sorting for the experimental evaluation of LPOD. The third workflow used for this evaluation is a real workflow, the Diagnosis Recommendation workflow [6] that was designed to predict a diagnosis label of patients for the health care domain. Our performance evaluation involves comparison with two other sate-of-the-art scheduling algorithms, IC-PCP [19] and genetic algorithm [23]. We assume an IaaS cloud environment customized from Amazon AWS with three types of Amazon EC2: T2.micro, T2.large and T2.xlarge of incremental computing capacities. In order to handle the performance fluctuation of task execution, we define an evaluation metric, C score, which is a slight improved version of the Fitness Score introduced in [24]. The definition of C is as follows:

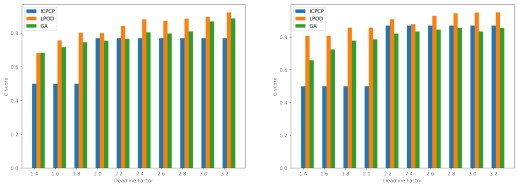
$$C = \begin{cases} 0.5 + 0.5 * \frac{maxcost - cost}{maxcost - mincost}, & if\ makespan \leq \delta \\ 0.5 - 0.5 * \frac{makespan - \delta}{maxmakespan - \delta}, & otherwise \end{cases} \quad (18)$$

where $maxcost$ is the monetary cost of running each task of the whole workflow on one instance of the most expensive VM type; $mincost$ is the monetary cost of running the whole workflow on one instance of the least expensive VM type and $maxmakespan$ is the makespan of running each task of the whole workflow on an instance of the slowest VM type.



(a) 60 seconds billing cycle for Montage. (b) 200 seconds billing cycle for Montage.

Fig. 7: C score for Montage workflow with 10M data size.



(a) 60 seconds billing cycle for Montage. (b) 200 seconds billing cycle for Montage.

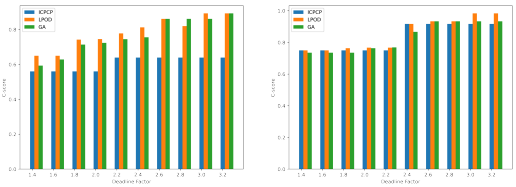
Fig. 8: C score for Montage workflow with 50M data size.

Before formally starting the experiment, the workflow schedule generated by LPOD for each workflow was submitted to the Workflow Executor in the DATAVIEW system to generate their provenance graphs. The configuration file records the execution time of tasks on their assigned EC2 VM instances (of three different VM types) and the data transfer time between consecutive tasks. The workflow input data size and the billing cycle period are two important factors in the process of workflow scheduling and the measure of workflow scheduling performance. Most commercial clouds, such as Amazon, charge users by hours. For this experiment, we applied the following two down-scaled billing cycles: a long billing cycle of $200s$ and a short billing cycle of $60s$. The deadline is another vital factor affecting the final result of workflow scheduling. We denote the actual finish time of the first partial path of each workflow running on the fastest VM without considering the data transfer time as M_f and apply the following rules in generating varied deadlines for the test workflows:

$$Deadline\ \delta = \lambda \times M_f \quad (19)$$

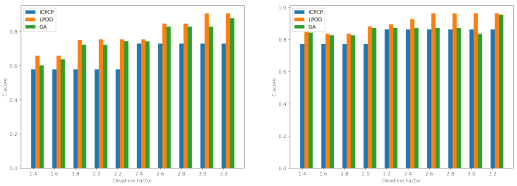
The synthesised tasks in the Montage and LIGO workflows are CPU sensitive, but the execution time of tasks in the Diagnosis Recommendation workflow on different VM types varies little in the three VM types above. So, λ is set to range from 1.4 to 3.2 with a step length of 0.2 for the Montage and LIGO workflows, and λ is set to range from 1.2 to 2.0 with a step length of 0.2 for the Diagnosis Recommendation workflow. There are 28 tasks in the Montage workflow, 26 tasks in the LIGO workflow, and 5 tasks in the Diagnosis Recommendation workflow. A small workflow input data size, $10M$, and a big workflow input data size, $50M$, were applied to the Montage workflow and LIGO workflow. Since the tasks in the Diagnosis Recommendation workflow are more computation demanding than the tasks in the other two workflows, a big workflow input data size, $20M$, was applied to the Diagnosis Recommendation workflow to avoid the task execution failure on VM instances.

All experiment results, as shown in Fig. 7, Fig. 8, Fig. 9, Fig. 10, Fig. 11, and Fig. 12, are focus on the C scores



(a) 60 seconds billing cycle for LIGO. (b) 200 seconds billing cycle for LIGO.

Fig. 9: C score for LIGO workflow with 10M data size.



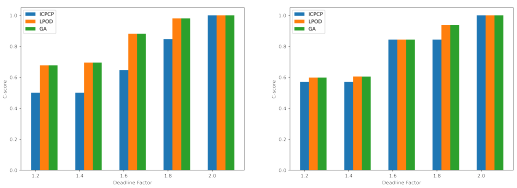
(a) 60 seconds billing cycle for LIGO. (b) 200 seconds billing cycle for LIGO.

Fig. 10: C score for LIGO workflow with 50M data size.

of the executed workflows per their respective workflow schedules generated by the three scheduling algorithms under different input data sizes, different deadlines and different billing cycles. A common observation is that, given each workflow, all the algorithms can successfully generate a workflow schedule and have the workflow finished before the deadline in the DATAVIEW system, even with a tight deadline constraint. The results show that in 95% cases a smaller billing cycle leads to higher C scores with other factors fixed. This is because when smaller billing period is applied, less idle time remains in VM instances, which causes a higher C score.

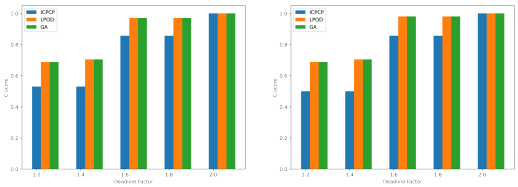
For the workflow schedules generated by the GA, we set the population size as 500 for a fixed iteration number as 500 with crossover probability as 0.8 and mutation probability as 0.01. Without fine-tuned parameters in the GA, the performance of GA surpasses LPOD in only two cases of the LIGO workflow with better random initial population. It is noticeable that the schedule generation time by GA is in second magnitude while LPOD and IC-PCP generate a schedule in million second magnitude.

Fig. 7 and Fig. 8 show that LPOD generates more efficient schedule than IC-PCP especially when deadline factor is less than 2.0, with highest increased C score as 21%



(a) 60 seconds billing cycle for Diagnosis Recommendation. (b) 200 seconds billing cycle for Diagnosis Recommendation.

Fig. 11: C score for Diagnosis Recommendation workflow with 10M data size.



(a) 60 seconds billing cycle for Diagnosis Recommendation. (b) 200 seconds billing cycle for Diagnosis Recommendation.

Fig. 12: C score for Diagnosis Recommendation workflow with 50M data size.

compared to IC-PCP's. Fig. 9 and Fig. 10 show that LPOD generate more efficient schedule than IC-PCP especially when deadline factor is bigger, with highest increased C score as 15% higher than IC-PCP's. Because the structure of the Diagnosis Recommendation workflow is linear, the three scheduling algorithms can generate the same schedule when deadline factor is 2.0, which assigns every task to the same cheapest VM instance as shown in Fig. 11 and Fig. 12. As the deadline increases, LPOD tends to generate the optimal schedule with resort to a set of VM instances of *different VM types* to optimize the assignment of the longest partial critical path. That explains why the C score increases as the deadline increases in most cases. It is also noticed that it is not always true that when the deadline factor increases, the C score is also increasing. This is because the LPOD is a local path optimized workflow scheduling. The first optimized path assignment may affect other path assignment which may increase the total accumulated cost. When the same billing cycle is applied with other parameter unchanged, the bigger input data size results in a higher C score, which are visualized in each pair results of the same workflow under the same billing cycle. This is because when the input size is bigger, the task execution time of each task is bigger. When the same billing period is applied, the idle time proportion of VM instances are smaller, which causes a higher C score.

In conclusion, the proposed LPOD scheduling algorithm consistently outperforms its two competitors in almost all cases tested — not only with the linear workflow structure of the Diagnosis Recommendation workflow but also with the complex workflow structures of the LIGO and Montage workflows.

5 Related Work

Workflow scheduling is a well-known complex research problem. Diverse algorithms, based on heuristic and meta-heuristic strategies, have been studied for the workflow scheduling problem in the cloud computing environment with different kinds of QoS requirements. Workflow scheduling methods have been classified into different categories in several comprehensive reviews [25] [26] [27].

Heterogeneous Earliest Finish Time (HETF) [28] assigns each task a priority value based on communication and execution cost, on which a greedy method is used to select the highest priority task and schedules it to the processor which process the shortest time. It is demonstrated that HEFT is not always successful when scheduling complex workflows

[29]. Several improved algorithms based on HEFT have been proposed to utilize a two-stage procedure to minimize the execution cost and the makespan of workflows simultaneously [30]. However, they do not consider a VM instance's booting time and the actual data transfer time between consecutive tasks. These algorithm selects the final scheduling solution from K identified best solutions. However, how to decide the best value for K is left unaddressed.

Critical paths have been utilized in the workflow scheduling problem with different kinds of heuristics [31] [32] [28]. Saeid et al. [19] proposed a deadline-constrained workflow scheduling algorithm that assigns a whole partial critical path to a single VM instance in order to minimize the data transfer time between consecutive tasks intuitively. This algorithm does not explore the possibility of assigning the tasks from one path to multiple VM instances in search for potentially better schedule solutions. Another flaw of this algorithm is that it does not take into account the transfer time of output data from a provisioned VM instance to its local storage for the child tasks running on the same VM instance when scheduling the deprovisioning of the VM instance. This arrangement is impractical for executing real-world workflows. Vahid et al. [29] modified the rank computation by composing a constrained critical path by incorporating the *in* and *out* degree information for a task, which gives a greater chance to execute ready tasks.

Meta-heuristic based approaches such as particle swarm optimization (PSO) [33] [34], genetic algorithms (GA) [23] and Ant Colony Optimization (ACO) [35], which finds feasible solutions in a large searching space. Such approaches can address workflow scheduling problem with multiple constrains easily, but they are time consuming and very expensive to generate a feasible schedule because of the initialization phase and huge solution space. It also takes time to tune different parameters.

Jyoti and Deo Prakash [36] proposed a just-in-time scheduling strategy that starts with a pre-processing step that combines pipelined tasks into a single task and then assigns such combined mega task to a single VM instance. Different from most other workflow scheduling algorithms in which each task is only visited once, this algorithm needs to visit each task twice (in both the pre-processing stage and the schedule generation stage). Their method may result in a schedule less cost-efficient than a mapping scheme that assigns pipelined tasks to multiple VM instances.

These scheduling methods assume that the necessary parameters to schedule a workflow is already provided, which incurs extra configuration effort in a MWfMS. The authors in [37] [38] used generated provenance data to dynamically tune the grouped tasks size on each VM to balance the workload instead of using the parameters fed in the workflow scheduling stage.

6 Conclusions and Future Works

In this paper, we systematically presented our innovative workflow scheduling algorithm, LPOD. Given any workflow, in order to generate an optimized schedule for the workflow, LPOD exploits the workflow parameters automatically curated from the provenance graph/data generated from prior (initial or tentative) execution of the workflow in the DATAVIEW system. The strategic steps incorporated in LPOD include: first, identify all the partial critical paths in an input workflow; second, try to schedule as many tasks in the prefix of each critical path to an existing VM instance as possible; third, resort to dynamic programming to explore *all possible mappings* of every task in the (unprocessed) suffix of each critical path to *any* feasible VM instance, resulting in an optimal schedule of the suffix of each path to the most suitable VM instances in a cloud environment. Comparing with all related works, our unique way of handling the suffixes of the critical paths is the most distinguishing part of the LPOD scheduling approach. The correctness and performance of LPOD have been experimentally evaluated based on both synthesised (simulating real workflows) and real-world workflow. As part of our near future work, we plan to extend the execution platform of DATAVIEW to other feature-rich environments, e.g., including secure and GPU VM instances.

Acknowledgement

This work is partially supported by National Science Foundation under grants CNS-1747095 and OAC-1738929.

References

[1] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman, "Experiences using cloud computing for a scientific workflow application," in *Proceedings of the 2nd international workshop on Scientific cloud computing*. ACM, 2011, pp. 15–24.

[2] M. Abouelhoda, S. A. Issa, and M. Ghanem, "Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support," *BMC bioinformatics*, vol. 13, no. 1, p. 77, 2012.

[3] V. C. Emeakaroha, M. Maurer, P. Stern, P. P. Łabaj, I. Brandic, and D. P. Kreil, "Managing and optimizing bioinformatics workflows for data analysis in clouds," *Journal of grid computing*, vol. 11, no. 3, pp. 407–428, 2013.

[4] M. Atkinson, C. S. Liew, M. Galea, P. Martin, A. Krause, A. Mouat, O. Corcho, and D. Snelling, "Data-intensive architecture for scientific knowledge discovery," *Distributed and Parallel Databases*, vol. 30, no. 5-6, pp. 307–324, 2012.

[5] F. Bhuyan, S. Lu, I. Ahmed, and J. Zhang, "Predicting efficacy of therapeutic services for autism spectrum disorder using scientific workflows," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 3847–3856.

[6] I. Ahmed, S. Lu, C. Bai, and F. A. Bhuyan, "Diagnosis recommendation using machine learning scientific workflows," in *2018 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2018, pp. 82–90.

[7] Z. Li, C. Yang, B. Jin, M. Yu, K. Liu, M. Sun, and M. Zhan, "Enabling big geoscience data analytics with a cloud-based, MapReduce-enabled and service-oriented workflow framework," *PloS one*, vol. 10, no. 3, 2015.

[8] A. Kashlev, S. Lu, and A. Mohan, "Big data workflows: A reference architecture and the dataview system," *Services Transactions on Big Data (STBD)*, vol. 4, no. 1, pp. 1–19, 2017.

[9] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *2008 grid computing environments workshop*. Ieee, 2008, pp. 1–10.

[10] "Amazon ec2 instances," 2020. [Online]. Available: https://www.ec2instances.info/

[11] J. Liu, E. Pacitti, P. Valduriez, D. De Oliveira, and M. Mattoso, "Multi-objective scheduling of scientific workflows in multisite clouds," *Future Generation Computer Systems*, vol. 63, pp. 76–95, 2016.

[12] H. Arabnejad and J. G. Barbosa, "Multi-QoS constrained and profit-aware scheduling approach for concurrent workflows on heterogeneous systems," *Future Generation Computer Systems*, vol. 68, pp. 211–221, 2017.

[13] M. R. Garey and D. S. Johnson, *Computers and intractability*. freeman San Francisco, 1979, vol. 174.

[14] M. Ebrahimi, A. Mohan, and S. Lu, "Scheduling big data workflows in the cloud under deadline constraints," in *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2018, pp. 33–40.

[15] A. Mohan, M. Ebrahimi, S. Lu, and A. Kotov, "Scheduling big data workflows in the cloud under budget constraints," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 2775–2784.

[16] A. Verma and S. Kaushal, "Deadline constraint heuristic-based genetic algorithm for workflow scheduling in cloud," *International Journal of Grid and Utility Computing*, vol. 5, no. 2, pp. 96–106, 2014.

[17] M. Mao and M. Humphrey, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 67–78.

[18] C. Bai, S. Lu, I. Ahmed, D. Che, and A. Mohan, "Lpod: A local path based optimized scheduling algorithm for deadline-constrained big data workflows in the cloud," in *2019 IEEE International Congress on Big Data (BigDataCongress)*. IEEE, 2019, pp. 35–44.

[19] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[20] I. Ahmed, S. Mofrad, S. Lu, C. Bai, F. Zhang, D. Che, and F. A. Bhuyan, "SGX-E2C2D: A Big Data Workflow Scheduling Algorithm for Confidential Cloud Computing," University of Wayne State, Department of Computer Science, Tech. Rep., 03 2019.

[21] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Ieee, 2008, pp. 1–12.

[22] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis," in *Workflows for e-Science*. Springer, 2007, pp. 39–59.

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

[24] S. Z. M. Mojab, M. Ebrahimi, R. G. Reynolds, and S. Lu, "iCATS: Scheduling big data workflows in the cloud using cultural algorithms," in *2019 IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2019.

[25] J. Liu, S. Lu, and D. Che, "A survey of modern scientific workflow scheduling algorithms and systems in the era of big data," in *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 2020, pp. 132–141.

[26] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, vol. 13, no. 4, pp. 457–493, 2015.

[27] M. Masdari, S. ValiKardan, Z. Shahi, and S. I. Azar, "Towards workflow scheduling in cloud computing: a comprehensive analysis," *Journal of Network and Computer Applications*, vol. 66, pp. 64–82, 2016.

[28] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[29] V. Arabnejad, K. Bubendorfer, B. Ng, and K. Chard, "A deadline constrained critical path heuristic for cost-effectively scheduling workflows," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2015, pp. 242–250.

[30] X. Zhou, G. Zhang, J. Sun, J. Zhou, T. Wei, and S. Hu, "Minimizing cost and makespan for workflow scheduling in cloud using fuzzy dominance sort based HEFT," *Future Generation Computer Systems*, vol. 93, pp. 278–289, 2019.

[31] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE transactions on parallel and distributed systems*, vol. 7, no. 5, pp. 506–521, 1996.

[32] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE transactions on Parallel and Distributed systems*, vol. 4, no. 2, pp. 175–187, 1993.

[33] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *2010 24th IEEE international conference on advanced information networking and applications*. IEEE, 2010, pp. 400–407.

[34] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang, "A market-oriented hierarchical scheduling strategy in cloud workflow systems," *The Journal of Supercomputing*, vol. 63, no. 1, pp. 256–293, 2013.

[35] W.-N. Chen and J. Zhang, "An ant colony optimization approach to a grid workflow scheduling problem with various qos requirements," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 39, no. 1, pp. 29–43, 2008.

[36] J. Sahni and D. P. Vidyarthi, "A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 2–18, 2018.

[37] D. de Oliveira, K. A. Ocaña, F. Baião, and M. Mattoso, "A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds," *Journal of grid Computing*, vol. 10, no. 3, pp. 521–552, 2012.

[38] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "Scientific workflow scheduling with provenance support in multisite cloud," in *International Conference on Vector and Parallel Processing*. Springer, 2016, pp. 206–219.

Changxin Bai is currently working toward the PhD degree in the Department of Computer Science at Wayne State University. He has received his M.S. in Computer Science from Wayne State University. He is a member of the Big Data Research Laboratory under the supervision of Dr. Shiyong Lu. His current research includes scientific workflows, big data workflows, provenance and data science.

Shiyong Lu, Ph.D., is a Professor of Department of Computer Science at Wayne State University. His current research interests focuses on scientific workflows and provenance data management. He has published 2 books and over 100 papers. He is the founding chair and program chair of IEEE International Workshop on Scientific Workflows since 2007. Dr. Lu is an associate editor of the International Journal of Cloud Applications and Computing, and an editorial board member of the International Journal of Healthcare Information Systems and Informatics. He is a Senior Member of the IEEE. He can be reached at shiyong@wayne.edu.

Dunren Che, Ph.D., is a Professor and the Program Director in the Department of Computer Science, Southern Illinois University Carbondale. His current research interests include Big Data (management and mining), Cloud Computing, Crowdsourcing, and Scientific Workflow Systems. He is a Senior Member of IEEE; he has published (authored/co-authored) more than 100 research papers.

Junwen Liu, Ph.D., graduated from the department of Computer Science at Wayne State University. He received his M.S. in Computer Science from Wayne State University. He was a member of the Big Data Research Laboratory under the supervision of Dr. Shiyong Lu. His current research interests include deep learning workflows in SWFMSs and their applications.