

# Schedule Compaction and Deadline Constrained DAG Scheduling for IaaS Cloud

Fuhui Wu<sup>(✉)</sup>, Qingbo Wu, Yusong Tan, Wei Wang, and Xiaoli Sun

College of Computer, National University of Defense Technology,  
410073 Changsha, China  
{fuhui.wu, qingbo.wu, yusong.tan, wei.wang, xiaoli.sun}@nudt.edu.cn

**Abstract.** Most cloud workflow scheduling algorithms assume that resources are charged under an ideal pay-as-you-go model, which may not be the case in real production cloud systems. Currently, most IaaS cloud providers charge users on billing cycle basis. If a resource is terminated before one billing cycle, the payment is still rounded up to one cycle. To address this problem, we firstly formalized it using bin-package method. Then, we propose a DAG schedule compaction algorithm of IC- $\star$ -SC, which compacts schedules generated by already exist algorithms to reduce resource requirement. Based on the compaction idea, we also propose a deadline constrained DAG scheduling algorithm of IC-SC. We compare our algorithms with state-of-the-art algorithms of IC-PCP and IC-PCPD2, and use 2 well-known scientific workflow applications for evaluation. Experimental results show that our algorithms reduce monetary cost drastically.

**Keywords:** Schedule compaction · IaaS cloud · Workflow scheduling · Deadline

## 1 Introduction

Many applications consist of a number of cooperative tasks which typically require more computing power beyond single machine capability. An easy and popular way is to describe complex applications using high-level representation in workflow method. As a single workflow can contain hundreds or thousands of tasks, the ever-growing data and computing requirements of workflows demand a higher-performance computing environment in order to execute the workflow in reasonable amount of time.

Traditional HPC systems are mostly dedicated and statically partitioned per administration policy. Owning a HPC system is inefficient in adapting to the surge of resource demand. With the development of cloud computing, workflow applications find a new way to solve this problem. Cloud computing targets at providing computing as service. With the IaaS cloud becoming mature, the long dream of providing computing service as the 5th utility [1] (after water, electricity, gas, and telephony) is gradually turning into reality.

In IaaS cloud environment, resources are served in a pay-per-use model and provisioned dynamically. Therefore, resources are provisioned on demand. To satisfy different QoS requirement of users, resource providers offer heterogeneous resources with various processing capabilities and prices. Usually, fast resource costs more money than that of slow resource. Thus, the cost/time trade-off problem becomes a hot topic in the literature. Ideally, users want to run their applications as fast as possible with minimum cost. If applications are not time-critical, a little delay can be tolerated for cost saving.

There are already algorithms concerning both time and monetary cost for cloud computing environment. However, most of them thought that leased resources can be completely utilized, and they are charged in an ideal pay-as-you-go model. This is not the case in real production system. As the Amazon EC2 cloud for example, they charge resources by hour. If a resource is terminated before one hour, the cost is still rounded up to one hour.

## 2 Related Works

The workflow scheduling problem has been studied extensively over the years. Kwok et al. [2] and Yu et al. [3] surveyed workflow scheduling on Multiprocessor system and distributed environments respectively. And Wicczorek et al. [4] focused on the taxonomies of the multi-criteria grid workflow scheduling algorithms. Most of them focus on decreasing the schedule length, which are classified as best-effort scheduling.

In contrast to best-effort scheduling, QoS constrained workflow scheduling is sometimes more close to that of real world application's requirement. In [5], Chen et al. studied the workflow scheduling problem with various QoS requirements and presented an Ant Colony Optimization (ACO) based approach. The target of their algorithm is to find a schedule that meets all QoS constraints and optimizes the user-preferred QoS objective.

Among all QoS constrained workflow scheduling problems in service oriented environment, deadline-constrained scheduling is one primary problem in the literature. DTL [6], DBL [7], and DET [8] are three heuristics that solve the deadline constrained workflow scheduling problem. Their basic idea is to distribute deadline over task partitions. Abrishami et al. [9,10] presented *Partial Critical Path* based scheduling algorithms, which distribute deadline in PCP-wise manner.

**Motivation:** To the best of our knowledge, there are few researches concerning about improving utilization of free time slots generated by task dependencies and billing cycle charged resources. In [11], they proposed a two-stage schedule compaction for duplication-based DAG scheduling. However, their target is to reduce resource number, which can not be directly applied to IaaS cloud system. In this paper, we address this highly unexplored aspect of DAG scheduling in IaaS cloud, and proposed a schedule compaction algorithm and a deadline constrained DAG scheduling algorithm in the following sections.

### 3 System Model and Definitions

#### 3.1 System Model

A workflow is modeled by a directed acyclic graph (DAG):  $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ , where  $\mathcal{T}$  consists of a set of tasks  $\{t_1, t_2, \dots, t_{|\mathcal{T}|}\}$  and  $\mathcal{E}$  consists of a set of directed edges  $\{e_i^j | (t_i, t_j) \in \mathcal{E}\}$ . Each task  $t_i$  is associated with a certain amount of computation workload  $wl_i$ . And each edge  $e_i^j$  carries  $d_i^j$  size of data send from task  $t_i$  to task  $t_j$ . Given a task graph, a task without any parent is called an *entry* task, and a task without any child is called an *exit* task. Without lose of generality, two dummy tasks  $t_{entry}$  and  $t_{exit}$  are added to the begin and end of  $\mathcal{G}$ . These dummy tasks have zero computation workload and zero-data dependencies to the actual *entry* or from the *exit* tasks.

There are theoretical unlimited resources provisioned by cloud providers and charged on a pay-as-you-go basis. The service provider offers several computation resource types  $\mathcal{RT} = \{rt_1, rt_2, \dots, rt_{|\mathcal{RT}|}\}$  to satisfy various QoS requirements of users. Each resource type  $rt_k$  is associated with a two-tuple information of  $\langle ECU, price \rangle$ , where  $ECU$  denotes the processing capability, and  $price$  denotes the monetary cost per charging unit. In general, scheduling task on faster resource costs more. In this paper, we assume a single cloud, where all resources are connected by a homogeneous network. The communication startup time (delay)  $L$  for each resource and communication bandwidth  $BW$  between any two resources are the same.

Hence, the execution time of a task on a resource is calculated as:  $ET_{t_i}^{rt_l} = ET_{t_i}^{rt(r_l)} = \frac{wl_i}{rt(r_l).ECU}$ ,<sup>1</sup> the data transmission time from task  $t_i$  to task  $t_j$  is calculated as:  $TT_{t_i}^{t_j} = L + \frac{d_i^j}{BW}$ . If two tasks are assigned to a same resource, the transmission time is zeroed. The monetary cost of network transmission is not considered in this paper. This assumption is reasonable as most popular commercial IaaS cloud providers charge users according to their leasing times for computing resources only.

#### 3.2 Schedule Definition

Based on this system model, a schedule is defined as:  $S = \langle \mathcal{R}, \mathcal{M}, makespan, cost \rangle$ .  $\mathcal{R} = \{r_1, r_2, \dots, r_{|\mathcal{R}|}\}$  is a set of used resources.  $\mathcal{M}$  consists of all mappings of task to resource, with  $m_{t_i}^{r_l} = \langle t_i, r_l, ST_{t_i}^{r_l}, FT_{t_i}^{r_l} \rangle$ , where  $ST_{t_i}^{r_l}$  is the start time of task  $t_i$  on resource  $r_l$  and  $FT_{t_i}^{r_l}$  is the finish time of task  $t_i$  on resource  $r_l$ . Then, the start time  $st(r_l)$  and termination time  $ft(r_l)$  of a resource  $r_l$  are calculated as:  $st(r_l) = \min_{m_{t_i}^{r_l} \in \mathcal{M}} \{ST_{t_i}^{r_l}\}$ ,  $ft(r_l) = \max_{m_{t_i}^{r_l} \in \mathcal{M}} \{FT_{t_i}^{r_l}\}$ . And, the rounded up lease time  $lt(r_l)$  for  $r_l$  is set with the minimal integer multiples of billing cycle length  $\tau$ :  $lt(r_l) = \lceil \frac{ft(r_l) - st(r_l)}{\tau} \rceil \times \tau$ .

<sup>1</sup>  $rt(r_l)$  is the resource type of resource  $r_l$ .

For a generated schedule  $S$ , the *makespan* is the overall schedule length. And the total cost  $C$  is the overall monetary payment. They are calculated as:  $makespan = \max_{m_{t_i}^{r_l} \in \mathcal{M}} \{FT_{t_i}^{r_l}\}$ ,  $C = \sum_{l=1}^{l=|\mathcal{R}|} \frac{lt(r_l)}{\tau} \times rt(r_l).price$ .

### 3.3 Bin-Package Problem Formalization

In this paper, we take the DAG scheduling problem in IaaS cloud as a bin-package problem, where each resource is taken as a bin, and tasks of DAG are taken as materials to be packed. However, the times consumed by one task when assigned to various resource types are different, which makes it a non-traditional bin-package problem. In order to explain our algorithms, we first make some definitions as following:

**Definition 1.** All tasks in  $\mathcal{T}$  are divided into two parts of already assigned tasks  $\mathcal{T}_a$  and unassigned tasks  $\mathcal{T}_u$ :  $\mathcal{T} = \mathcal{T}_a \cup \mathcal{T}_u$ .

**Definition 2.**  $\mathcal{T}_{r_l}$ : It contains all tasks that are assigned to resource  $r_l$ . An execution order is imposed on all tasks of  $\mathcal{T}_{r_l}$ , and a position  $pos(t_i)$ , starting from 0, is assigned for each task  $t_i$ .

**Definition 3.**  $EST(t_i)$ ,  $EFT(t_i)$ : The earliest start time of task  $t_i$ , which is determined by its immediate parents and the pre-execution task  $t_k$  if exist.

$$EST(t_i) = \begin{cases} \max\{\max_{t_j \in \mathcal{P}(t_i)} \{EFT(t_j) + TT_{t_j}^{t_i}\}, EFT(t_k)\}, & \exists t_k : pos(t_i) = pos(t_k) + 1 \\ \max_{t_j \in \mathcal{P}(t_i)} \{EFT(t_j) + TT_{t_j}^{t_i}\}, & otherwise \end{cases} \quad (1)$$

$$EFT(t_i) = EST(t_i) + ET(t_i) \quad (2)$$

$\mathcal{P}(t_i)$  contains all the immediate predecessors of task  $t_i$ . Assuming  $\mathcal{RT}.fastest$  being the fastest resource type in  $\mathcal{RT}$ , then the  $ET$  is calculated as:

$$ET(t_j) = \begin{cases} ET_{t_j}^{r_l}, & \exists r_l : m_{t_j}^{r_l} \in \mathcal{M} \\ ET_{t_j}^{\mathcal{RT}.fastest}, & otherwise \end{cases} \quad (3)$$

**Definition 4.**  $ST(t_i)$ ,  $FT(t_i)$ : The start time and finish time of already assigned task  $t_i$ . They are set with the earliest start time and earliest finish time on mapped resource  $r_l$ . Hence, they can be changed (usually be delayed) with the updating of  $EST$  and  $EFT$ .

**Definition 5.**  $LFT(t_i)$ ,  $LST(t_i)$ : The latest finish time of task  $t_i$  is determined by its immediate children and position if it is assigned to resource.

$$LFT(t_i) = \begin{cases} \min\{\min_{t_j \in \mathcal{C}(t_i)} \{LST(t_j) - TT_{t_j}^{t_i}\}, LST(t_k)\}, & \exists t_k : pos(t_k) = pos(t_i) + 1 \\ \min\{\min_{t_j \in \mathcal{C}(t_i)} \{LST(t_j) - TT_{t_j}^{t_i}\}, st(r_l) + lt(r_l)\}, & pos(t_i) = |\mathcal{T}_{r_l}| - 1 \\ \min_{t_j \in \mathcal{C}(t_i)} \{LST(t_j) - TT_{t_j}^{t_i}\}, & otherwise \end{cases} \quad (4)$$

$$LST(t_i) = LFT(t_i) - ET(t_i) \quad (5)$$

$\mathcal{C}(t_i)$  contains all the immediate successors of  $t_i$ . To calculate the LFT of task  $t_i$ , the  $\min_{t_j \in \mathcal{C}(t_i)} \{LST(t_j) - TT_{t_j}^{t_i}\}$  part assures that its immediate children are able to start no late than the LST of them. If there is a task  $t_k$  assigned right behind it, it should finish no late than the LST of  $t_k$ . If it is the last task of assigned resource  $r_l$ , the LFT( $t_i$ ) should not exceed  $st(r_l) + lt(r_l)$ . Otherwise, the monetary cost of  $r_l$  will increase.

**Definition 6.** *Insert Operation:* A legal insert operation denotes a valid task-resource mapping for this special bin-package problem. To define a legal insert operation, we introduce a concept of slot first. A slot is a time fragment in a resource  $r_l$ . Assuming the position of slot in resource  $r_l$  is  $pos(slot)$ , the start time  $st(slot)$  and finish time  $ft(slot)$  are defined as:

1. **If**  $\mathcal{T}_{r_l} = NULL$ , meaning  $r_l$  is a new resource without any task assigned to it, **then**  $st(slot) = 0$ ,  $ft(slot) = +\infty$ , and  $pos(slot) = 0$ ;
2. **else**,
  - (a) **if**  $pos(slot) = 0$ , **then**  $st(slot) = \max\{LFT(t_i) - lt(r_l), 0\}$ , and  $ft(slot) = LST(t_j)$ , where  $pos(t_i) = |\mathcal{T}_{r_l}| - 1$  and  $pos(t_j) = pos(slot)$ ;
  - (b) **else if**  $pos(slot) > 0$  and  $pos(slot) < |\mathcal{T}_{r_l}|$ , **then**  $st(slot) = EFT(t_i)$ , and  $ft(slot) = LST(t_j)$ , where  $pos(t_i) = pos(slot) - 1$  and  $pos(t_j) = pos(slot)$ ;
  - (c) **else**  $pos(slot) = |\mathcal{T}_{r_l}|$ , **then**  $st(slot) = EFT(t_i)$  and  $ft(slot) = EST(t_j) + lt(r_l)$ , where  $pos(t_i) = |\mathcal{T}_{r_l}| - 1$  and  $pos(t_j) = 0$ .

Then, a task  $t_i$  is able to be inserted into a slot in resource  $r_l$ , subjecting to: (i)  $(\max\{EST(t_i), st(slot)\} + ET_{t_i}^{r_l}) \leq ft(slot)$ ; (ii)  $EFT(t_i) \leq LFT(t_i)$ .

**Definition 7.** *Append operation:* It is an extension in case 2(c) of insert operation, except that the  $ft(slot)$  is set  $+\infty$ . And, a legal append operation should satisfy the same conditions as insert operation.

## 4 IC-★-SC: DAG Schedule Compaction

In this section, we propose a DAG schedule compaction algorithm named IC-★-SC. The IC-★-SC algorithm itself doesn't assign tasks to resources. It works on schedules generated by already exist algorithms. The symbol of ★ in IC-★-SC denotes the algorithm that generates schedules to be compacted.

### 4.1 The Basis of IC-★-SC

The basic idea of IC-★-SC algorithm is to compact tasks, already assigned to resources, into fewer resources. As the resources leased from IaaS cloud are charged in billing cycle unit, the monetary cost can be reduced only if all tasks that cover a billing cycle are compacted to other leased resources. Hence, we propose a split-wise DAG schedule compaction algorithm.

To introduce the conception of *split*, all resources in  $\mathcal{R}$  are divided into two parts of *compacted resource* ( $\mathcal{R}_c$ ) and *scheduled resource* ( $\mathcal{R}_s$ ), where  $\mathcal{R}_s$  is initialized as  $\mathcal{R}$ , and  $\mathcal{R}_c$  is empty at the beginning. Each *split* is chosen only from  $\mathcal{R}_s$ . After compaction, all leased resources are in  $\mathcal{R}_c$  and  $\mathcal{R}_s$  is empty.

A *split* is a set of tasks covering one billing cycle of a resource in  $\mathcal{R}_s$ . Each resource  $r_l$  in  $\mathcal{R}_s$  is divided into  $nb\_split(r_l)$  *splits* in (6). An  $index(r_l)$  parameter is set for every resource  $r_l$  to record current *split* position. After compaction of one *split* in a resource, it is increased by one and point to the next *split*. The *split* start time and finish time are computed in (7) and (8).

$$nb\_split(r_l) = \lceil \frac{\max_{\forall t_i: m_{t_i}^{r_l} \in \mathcal{M}} \{LFT(t_i)\} - \min_{\forall t_i: m_{t_i}^{r_l} \in \mathcal{M}} \{EST(t_i)\}}{\tau} \rceil \quad (6)$$

$$st(split) = \max_{\forall t_i: m_{t_i}^{r_l} \in \mathcal{M}} \{LFT(t_i)\} - \tau \times (nb\_split(r_l) - index(r_l)) \quad (7)$$

$$ft(split) = st(split) + \tau \quad (8)$$

For each compaction process, a  $nextSplit()$  procedure is called to select a next *split*. There may be several *split* candidates from different resources in  $\mathcal{R}_s$ . The one with the smallest split start time will be selected. And the more expensive one is chosen if ties. After a *split* has been determined, corresponding tasks should be added to *split*. For a task  $t_i$  on selected  $r(split)$  (the resource where *split* from), it is added into *split* if the start time and finish time  $[ST_{t_i}^{r(split)}, FT_{t_i}^{r(split)}]$  intersects with  $(st(split), ft(split))$ .

## 4.2 The IC-★-SC Algorithm

The IC-★-SC algorithm consists of two main procedures of *latest time updating* and *split compaction*. The first procedure determines the *LSTs* and *LFTs* of tasks on resources they are mapped. The target is to increase intervals between neighbor tasks on the same resource, which produces larger free time *slots* for compaction. The second procedure achieves schedule compaction actually. Algorithm 1 describes the overview procedure of IC-★-SC algorithm.

For initialization, it sets the *LST* and *LFT* for the dummy exit task  $t_{exit}$ . The *LST* is set with a deadline, if required. Otherwise, it is set with the makespan of the original schedule. After that, the IC-★-SC iteratively calls *latest time updating* and *split compaction* procedures until all candidate *splits* are processed.

To compute the latest times of all tasks. It tries to move tasks as late as possible on mapped resources, which can be implemented using a up-wards breadth-first search method from the dummy exit task  $t_{exit}$ . The *LFT* of a task is calculated using (4–5).

## 4.3 Split Compaction Procedure

For a selected *split*, it iteratively tries to insert tasks in the *split* to other resources. A success insert operation is defined in Definition 6. The resources in  $\mathcal{R}_c$  are selected firstly. If the insertion fails, resources in  $\mathcal{R}_s$  are selected.

---

**Algorithm 1.** The IC- $\star$ -SC Algorithm

---

**Input:**  $S$ : the schedule to be compacted;  
        $makespan$ : the makespan of schedule  $S$ ;  
        $D$ : the deadline, if required, for  $\mathcal{G}$ .  
**Output:**  $S'$ : a new schedule after compaction of  $S$ .  
1: **if** deadline is required **then**  
2:    $LST(t_{exit}) = D$   
3: **else**  
4:    $LST(t_{exit}) = makespan$   
5: **end if**  
6: update the  $LST$ s and  $LFT$ s of all tasks in  $\mathcal{T}$ ;  
7: **while** has an unprocessed split **do**  
8:    $split \leftarrow nextSplit()$ ;  
9:   call for  $compact(split)$ ;  
10: **end while**

---

After the insertion attempt of all tasks in selected  $split$ , there are 2 cases. If all insertions succeed, all tasks in  $r(split)$  are separated into three parts according to their positions. Tasks in  $split$  are inserted to other resources. Tasks before  $split$  are kept in  $r(split)$  and the resource  $r(split)$  is moved to  $\mathcal{R}_c$ . Tasks after  $split$  are assigned to a new resource with the same resource type as  $r(split)$ , and this new resource is added to  $\mathcal{R}_s$ . However, if the insertion fails, it should roll the schedule back to the point before compaction of  $split$ .

---

**Algorithm 2.** The *compact* Procedure

---

**Input:**  $split$ : the split to be processed.  
**Output:**  $S_{tmp}$ : an intermediate schedule after compaction of  $split$ .  
1: **for all**  $t_i \in split$  **do**  
2:   insert  $t_i$  to the first-fit resource in  $R$ ;  
3:   **if** insertion succeeds **then**  
4:     iteratively update the  $EST$ s and  $EFT$ s of  $succ(t_i)$  and the immediate task following  $t_i$  on  $r(split)$ ;  
5:   **else**  
6:     **break**;  
7:   **end if**  
8: **end for**  
9: **if** all tasks in  $split$  are inserted into other resources **then**  
10:   (i) schedule all tasks, if exist, after  $split$  to a new resource  $r_k$  with the same type as  $r(split)$ , and add  $r_k$  to  $\mathcal{R}_s$ ;  
11:   (ii) move  $r(split)$ , if there are still tasks assigned to it, to  $\mathcal{R}_c$ ;  
12:   (iii) update the  $LST$ s and  $LFT$ s of all tasks in  $\mathcal{T}$ ;  
13: **else**  
14:   rollback  $S_{tmp}$  to the point before compaction of  $split$ ;  
15: **end if**

---

#### 4.4 Time Complexity

The most time consuming part of the overall IC-★-SC algorithm is the *updateLatestTime* and *compact* procedures. To update the *LFTs*, IC-★-SC adopts breadth-first traverse method from  $t_{exit}$ . It costs a time complexity of  $O(|\mathcal{T}| + |\mathcal{E}|)$ , which approximately equals to  $O(|\mathcal{T}|^2)$  for dense workflows with a edge between any two tasks. As the hole IC-★-SC algorithm compact all tasks in split-wise manner, and there are at most  $|\mathcal{T}|$  splits, the *updateLatestTime* contributes  $|\mathcal{T}|^3$  time complexity to the IC-★-SC algorithm.

The most time consuming part of the *compact* procedure is the insert operations for *split* tasks. To update the *ESTs* and *EFTs* for each insertion attempt at line 3 of Algorithm 2, the time complexity is  $O(|\mathcal{T}| + |\mathcal{E}|)$ , approximating to  $O(|\mathcal{T}|^2)$ , in the worst case. Overall, each task will be insert once. Hence, the time complexity for  $\mathcal{T}$  tasks of all *compact* procedure is  $O(|\mathcal{T}|^3)$ .

At last, the time complexity of IC-★-SC is  $O(|\mathcal{T}|^3)$ .

### 5 IC-SC: Deadline Constrained DAG Scheduling

To take advantage of the bin-package idea in IC-★-SC algorithm. We propose a deadline constrained DAG scheduling algorithm that implements the bin-package mechanism at scheduling time.

#### 5.1 IC-SC Algorithm

Deadline constrained DAG scheduling is a widely studied problem in the literature [6–10]. IC-PCP [9] and IC-PCPD2 [9] are state-of-the-art algorithms for billing cycle aware IaaS cloud environment. In order to introduce the IC-SC algorithm, we first make an review of the IC-PCP and IC-PCPD2 algorithms. The core conception of them is *partial critical path(PCP)* [9]. The details of *PCP* are not repeated here.

There are two main categories of workflow scheduling algorithms: *list* scheduling and *clustering* heuristics:

IC-PCP is a *clustering* heuristic. Two parts, that compose IC-PCP *clustering* heuristic, are:

1. *Clustering*. It determines how tasks are mapped to resources. IC-PCP maps all tasks in one *PCP* to the same resource to reduce data transmission times.
2. *Ordering*. It determines the execution order of all tasks on the same resource. The IC-PCP algorithm schedules all *PCP* tasks to a same resource, which is selected using a monetary cost optimization mechanism defined by them.

IC-PCPD2 is a *list* scheduling heuristic. It firstly assign subdeadlines to all tasks using the similar strategy to the IC-PCP algorithm. The actual scheduling is carried out in *list* scheduling manner, including two parts:

1. *Task selection*. It determines which task to schedule next. The IC-PCPD2 algorithm randomly selects a ready task.



2. *Resource selection.* It determines which resource the selected task is assigned to. The IC-PCPD2 algorithm always selects the cheapest resource, either an existing resource or a new one.

According to Abrishami et al., both IC-PCP and IC-PCPD2 have a promising performance, with IC-PCP performing better than IC-PCPD2 in most cases. However, the IC-PCP needs to schedule all tasks on *PCP* to the same resource. Moreover, the position of *PCP* task are limited on assigned resource. To utilize the bin-package idea of IC- $\star$ -SC, we propose a deadline constrained DAG scheduling heuristic, named IC-SC. It also schedules workflow tasks in *PCP* manner. But the IC-SC algorithm doesn't need all *PCP* tasks to be assigned to a same resource. Moreover, the position for *PCP* task is more free.

The main structure of IC-SC is the same as IC-PCP. We don't repeatedly explain them here. The most obvious difference with IC-PCP lies in scheduling of *PCP* tasks. The IC-SC algorithm schedules each task in current *PCP* separately. It first tries to schedule *PCP* task to an already exist resource. It uses the insertion mechanism defined by Definition 6. The cheapest already exist resource that is able to insert the scheduling *PCP* task is chosen. And the scheduling *PCP* task is inserted to the first-fit slot in the chosen resource. If there is not an already exist resource that is able to insert the scheduling *PCP* task, two cases are considered. First, it tries to append the scheduling task to an already exist resource. The cheapest one that is able to append is assumed to be  $r_l$ . When implementing the append operation, we make a restriction that it should not increase a complete free billing cycle. As there are already 3 billing cycles after rounded up for example, the 4th billing cycle should be utilized at least partly by the appended task. Otherwise, this append operation is illegal. Second, it tries to lease a new resource, assuming to be  $r'_l$ , and inserts the scheduling task to it. If  $r_l$  exists and  $r_l$  is cheaper than  $r'_l$ , the scheduling task is appended to  $r_l$ . Otherwise, it is inserted to a new resource  $r'_l$ . The reason behind the append operation is to utilize the last slot of an already exist resource and reduce communication cost.

## 5.2 Time Complexity

The first part of the algorithm is to initialize the parameters of each workflow task, which requires the same time complexity of  $O(|T|^2)$  as IC-PCP algorithm. Then the second part of the algorithm is recursively scheduling workflow tasks to resources. We consider its overall actions instead of entering into details. Overall, the Parents Assigning procedure schedules each workflow task only once, and updates the parameters of its successors and predecessors. To schedule a task, there are three cases in all. First, there are at most  $O(|T|)$  slots in all the already exist resources for insertion. If insertion fails, there are also at most  $O(|T|)$  resources for appending. If a new resource has to be started, there are constant number of available resource types. Hence, the time complexity to schedule  $|T|$  workflow tasks is  $O(|T|^2)$ . On the other hand, each task has at most  $(|T| - 1)$  successors and predecessors, the time complexity of the updating part for all workflow tasks is also  $O(|T|^2)$ . Consequently, the overall time complexity of the

Parents Assigning procedure is  $O(|T|^2)$ , which is also the time complexity of the IC-SC algorithm.

## 6 Performance Evaluation

### 6.1 The Initial Schedule Generating and Comparative Algorithms

We choose two state-of-the-art scheduling algorithms in the literature to generate initial schedules: IC-PCP and IC-PCPD2. Both of them are designed for billing cycle charged IaaS cloud. The schedules generated by them meet deadline constraint. To evaluate the performance of IC-SC algorithm, both IC-PCP and IC-PCPD2 are also taken as comparative algorithms.

### 6.2 Experimental Workflows

Figure 1(a) and (b) shows the approximate structures of selected workflows we choose from [12] to evaluate the performance of our algorithm. They are:

- Montage: astronomy;
- CyberShake: earthquake science.

They developed a workflow generator, which can create synthetic workflows. Using this workflow generator, they create different sizes for each workflow application in terms of total number of tasks. These workflows are available in DAX (Directed Acyclic Graph in XML) format from their website<sup>2</sup>, from which we choose 6 sizes (50, 100, 200, 300, 400, 500) for each workflow.

### 6.3 Experimental Setup

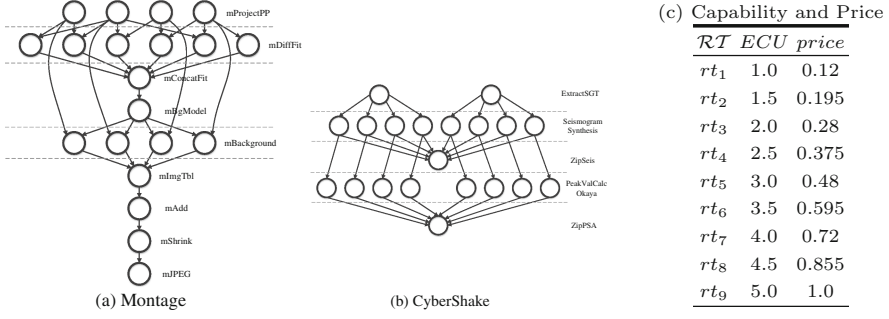
In our experiments, we assume a IaaS cloud environment offering 9 different computation services, with different processing capability and prices as shown in Fig. 1(c). They satisfy that slower resource has higher performance/cost ratio. The average bandwidth between computation resources is set to 20MBps.

An important parameter of the experiments is the billing cycle. To evaluate the impact of short and long billing cycles on our algorithms, we consider two different billing cycles in the experiments: a long one equal to one hour, and a short one equal to five minutes.

All costs are normalized by the cost of the schedule generate by HEFT [13] as baseline  $C_{base}$ . We implement a modified HEFT algorithm for IaaS cloud environment. It always chooses the resource that finish selected task earliest. And then, the normalized cost of a schedule is defined as:  $NC = \frac{C}{C_{base}}$ .

Finally, to evaluate the impact of slackness degree of a schedule on our compaction and scheduling algorithms, we need to assign a deadline to each workflow. We firstly define the fastest schedule for a workflow be the one get by HEFT [13]. The makespan of this schedule is denoted by  $M_F$ . And then, the deadline of a workflow is set to be  $\alpha \times M_F$ ,  $\alpha \in \{2.5, 3, 3.5, 4, 4.5\}$ .

<sup>2</sup> <http://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator>.

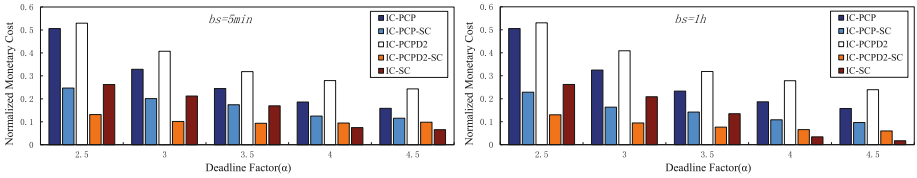


**Fig. 1.** The structures of benchmark scientific workflows, and information of available resource types.

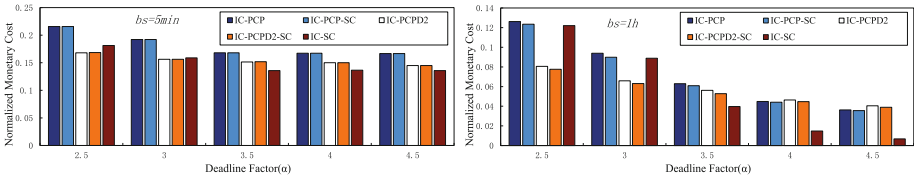
## 6.4 Experimental Results

Figures 2 and 3 show the average cost of scheduling workflows with IC-PCP, IC-PCPD2, IC-SC algorithms, the average cost after compaction on schedules generated by IC-PCP, IC-PCPD2 algorithms. The IC-PCP-SC algorithm compacts schedules generated by IC-PCP algorithm, and the IC-PCPD2-SC algorithm compacts schedules generated by IC-PCPD2 algorithm correspondingly.

We can get that IC- $\star$ -SC is able to reduce cost dramatically after compaction over schedules generated by IC-PCP and IC-PCPD2 algorithms. They also show that the IC-SC algorithm is able to generate less monetary cost schedules than IC-PCP and IC-PCPD2 algorithms. With looser deadline constraint, the schedules generated by IC-SC cost even less than IC-PCP-SC and IC-PCPD2-SC.



**Fig. 2.** Normalized costs of Montage application.



**Fig. 3.** Normalized costs of CyberShake application.

## 7 Conclusion

We developed a novel algorithm to reduce billing cycle charged resource requirement for schedules generated by any scheduling algorithm. When applied on a valid schedule, the proposed IC-★-SC algorithm compacts the schedule to fewer number of resources without increasing the schedule length or under a deadline constraint. We also proposed an efficient deadline constrained workflow scheduling algorithm for IaaS cloud. The time complexity is small, which makes it suitable for large scale workflow scheduling. Experiments on scientific workflows verified that IC-★-SC algorithm dramatically reduces the resource requirement of the schedules generated by IC-PCP and IC-PCPD2 algorithms. And, IC-SC performs better than the state-of-the-art algorithms, especial for workflows with loose deadline constraint.

**Acknowledgments.** This work is supported by project (2013AA01A212) from the National 863 Program of China, project (61202121) from the National Natural Science Foundation of China, Science and technology project (2013Y2-00043) in Guangzhou of China.

## References

1. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.* **25**(6), 599–616 (2009)
2. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv. (CSUR)* **31**(4), 406–471 (1999)
3. Yu, J., Buyya, R., Ramamohanarao, K.: Workflow scheduling algorithms for grid computing. In: Xhafa, F., Abraham, A. (eds.) *Metaheuristics for Scheduling in Distributed Computing Environments*, pp. 173–214. Springer, Heidelberg (2008)
4. Wiczorek, M., Hoheisel, A., Prodan, R.: Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Gener. Comput. Syst.* **25**(3), 237–256 (2009)
5. Chen, W.-N., Zhang, J.: An ant colony optimization approach to a grid workflow scheduling problem with various QoS requirements. *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.* **39**(1), 29–43 (2009)
6. Yu, J., Buyya, R., Tham, C.K.: Cost-based scheduling of scientific workflow applications on utility grids. In: *First International Conference on e-Science and Grid Computing*, p. 8. IEEE (2005)
7. Yuan, Y., Li, X., Wang, Q., Zhang, Y.: Bottom level based heuristic for workflow scheduling in grids. *Chin. J. Comput.* **31**(2), 282 (2008)
8. Yuan, Y., Li, X., Wang, Q., Zhu, X.: Deadline division-based heuristic for cost optimization in workflow scheduling. *Inf. Sci.* **179**(15), 2562–2575 (2009)
9. Abrishami, S., Naghibzadeh, M., Epema, D.H.: Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Gener. Comput. Syst.* **29**(1), 158–169 (2013)
10. Abrishami, S., Naghibzadeh, M., Epema, D.H.: Cost-driven scheduling of grid workflows using partial critical paths. *IEEE Trans. Parallel Distrib. Syst.* **23**(8), 1400–1414 (2012)

11. Bozdag, D., Ozguner, F., Catalyurek, U.V.: Compaction of schedules and a two-stage approach for duplication-based DAG scheduling. *IEEE Trans. Parallel Distrib. Syst.* **20**(6), 857–871 (2009)
12. Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M. H., Vahi, K.: Characterization of scientific workflows. In: 2008 Third Workshop on Workflows in Support of Large-Scale Science, pp. 1–10. IEEE, November 2008
13. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.* **13**(3), 260–274 (2002)

Cloud Computing and Big Data

Second International Conference, CloudCom-Asia 2015,

Huangshan, China, June 17-19, 2015, Revised Selected

Papers

Qiang, W.; Zheng, X.; Hsu, C.-H. (Eds.)

2015, XVII, 400 p. 161 illus. in color., Softcover

ISBN: 978-3-319-28429-3