Future Generation Computer Systems 27 (2011) 1011–1026

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Cost optimized provisioning of elastic resources for application workflows

Eun-Kyu Byun [a], Yang-Suk Kee [b], Jin-Soo Kim [c,*], Seungryoul Maeng [a]

[a] Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon 305-701, South Korea

[b] Oracle USA Inc., Redwood Shores, CA 94065, USA

[c] School of Information and Communication Eng., Sungkyunkwan University, Suwon, Gyeonggi-do 440-746, South Korea

ARTICLE INFO

Article history:
Received 19 October 2010
Received in revised form
19 April 2011
Accepted 3 May 2011
Available online 10 May 2011

Keywords:
Resource capacity estimation
Resource allocation
Application workflow
Cloud computing economy

ABSTRACT

Workflow technologies have become a major vehicle for easy and efficient development of scientific applications. In the meantime, state-of-the-art resource provisioning technologies such as cloud computing enable users to acquire computing resources dynamically and elastically. A critical challenge in integrating workflow technologies with resource provisioning technologies is to determine the right amount of resources required for the execution of workflows in order to minimize the financial cost from the perspective of users and to maximize the resource utilization from the perspective of resource providers. This paper suggests an architecture for the automatic execution of large-scale workflow-based applications on dynamically and elastically provisioned computing resources. Especially, we focus on its core algorithm named PBTS (Partitioned Balanced Time Scheduling), which estimates the minimum number of computing hosts required to execute a workflow within a user-specified finish time. The PBTS algorithm is designed to fit both elastic resource provisioning models such as Amazon EC2 and malleable parallel application models such as MapReduce. The experimental results with a number of synthetic workflows and several real science workflows demonstrate that PBTS estimates the resource capacity close to the theoretical low bound.

1. Introduction

Many applications for large-scale scientific problems consist of a number of cooperative tasks which typically require more computing power beyond single machine capability. Distributed HPC (High Performance Computing) environments such as cluster, grid, and IaaS (Infrastructure as a Service) cloud have become a viable computing platform for this class of applications. An easy and popular way to describe complex applications is to use a high-level representation. Especially, many projects in a variety of disciplines adopt a workflow technology [1–4]. The MapReduce programming model from Google [5] and Dryad from Microsoft [6] are also an example of workflow approach. A workflow is represented as a Directed Acyclic Graph (DAG) with nodes and edges, which represent tasks and data/control dependencies between tasks, respectively. A workflow can specify the overall behavior and structure of an application independent of target execution environments. Then, the workflow management systems such as Pegasus [7], Askalon [8], and Triana [9] deal with the complexity of application management and allow scientists to execute the workflow on distributed resources.

Traditional HPC systems are mostly dedicated and statically partitioned per administration policy. Owning a HPC system, however, is not only very expensive but also inefficient in adapting to the surge of resource demand. For this reason, the dynamic coordination and provision of distributed resources rapidly draw attraction from scientists. Some notable achievements are the resource virtualization and provisioning technologies such as COD (Cluster on-demand) [10], Virtual Grid [11,12], Eucalyptus [13], and IaaS Cloud such as Amazon's EC2 (Elastic Compute Cloud) [14].

Workflow technology can benefit from resource provisioning technology. As observed in IaaS, resource provisioning technology makes the resource allocation process simple and straightforward; users only need to identify resource type, leasing period, and cost for their applications. Since resource types tend to depend on application characteristics, the main concerns of users would be the leasing period and the cost. Ideally, users want to run their applications as fast as possible with the minimum cost. However, if applications are not time-critical, users may tolerate a little delay of execution for more cost saving.

A critical issue in the integration of the workflow systems and the resource provisioning technologies is to determine the amount

* Corresponding author.
E-mail addresses: ekbyun@camars.kaist.ac.kr (E.-K. Byun), yang.seok.ki@oracle.com (Y.-S. Kee), jinsookim@skku.edu (J.-S. Kim), maeng@camars.kaist.ac.kr (S. Maeng).

0167-739X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2011.05.001

of resources that workflow systems should request to provisioning systems, which is termed *resource capacity* in this paper. The resource capacity affects the total execution time (makespan) of application workflow and determines the financial cost.

If the amount of resources is large enough, the makespan can be reduced by executing independent tasks simultaneously. However, too many resources can lead to low resource utilization, high scheduling overheads, and most importantly high cost. On the contrary, if the amount of resources is too small, the execution time of workflow can increase and in consequence the time constraints of applications cannot be satisfied.

To solve this problem, we proposed a heuristic algorithm named Balanced Time Scheduling (BTS) [15,16]. BTS estimates the minimum number of computing resources required to execute a workflow within a given deadline. We also built a prototype system to prove the concept by integrating Pegasus for workflow management and the Virtual Grid for resource management [17]. In BTS, the size of computing resources is kept static throughout the entire workflow execution. Even though BTS is cost-efficient, scalable, and generic, however, the static allocation strategy has a potential resource waste because BTS may not use the entire resources always for the leasing period.

On the contrary, state-of-the-art provisioning services allow users to elastically adjust the size of computing resources at runtime per workload. However, we notice that the smallest charge unit of resource lease for most providers is one hour. For example, Amazon EC2 [14] adopts an hour-based pay-as-you-go charge policy. Another example is that most clusters for Grid computing such as TeraGrid [18] have hour-based advance reservation policies. In addition, the startup and cleanup overhead, such as initializing and terminating virtual machines in the cloud and the resource mapping in batch systems, is not negligible. As such, fine-grained management for elastic resources is not practical yet.

In response, this paper proposes an algorithm named the Partitioned Balanced Time Scheduling (PBTS) algorithm, which determines the best number of computing resources per time charge unit in elastic computing environments, minimizing the gross cost during the entire application lifetime. Fundamentally, the PBTS algorithm is an extension of the BTS algorithm for elastic resources, and it inherits all the benefits of BTS. First, PBTS is a polynomial algorithm, which is scalable to very large workflows having tens of thousands of tasks and edges. Second, PBTS can handle workflows with data-parallel tasks and MPI-like parallel tasks whose subtasks are executed concurrently on distinct resources. Beyond the inherited characteristics, PBTS makes a full use of the elasticity of resources and provides the user with a resource allocation plan having the lowest cost. This is the unique contribution that conventional workflow scheduling techniques cannot achieve. PBTS is designed to adjust the resource allocation at runtime so that PBTS can find the resource capacity with lower cost when tasks are completed earlier than the predicted time.

The rest of this paper is organized as follows. Section 2 presents the architecture for workflow execution on elastic resources and defines the resource capacity estimation problem. In Section 3, we provide an overview of prior studies closely related to our work. Section 4 details the proposed PBTS algorithm. The methodology and the experimental results are presented in Section 5. Finally, Section 6 concludes the paper with future research directions.

2. Workflow execution on elastic resources

This section introduces the overall architecture of the computing framework to leverage elastic resources, and explains the models of application and resource for this study.

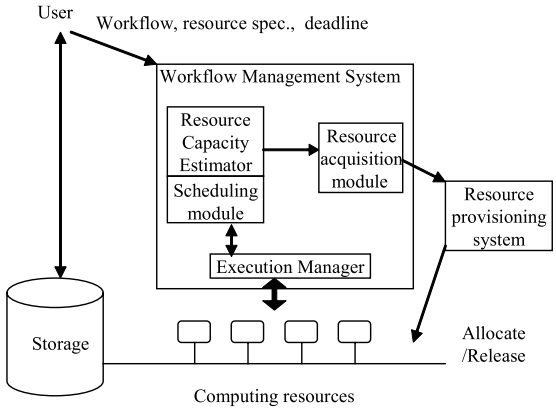


Fig. 1. Computing platform model.

2.1. Integrated workflow execution system

Fig. 1 depicts our computing platform model for executing workflows on elastic resources and a simple working scenario. We use the same computing platform model used in our prior study [15,16] except that resources are elastic with a discrete lease period. First, a user asks the *Workflow Management System (WMS)* to execute a workflow within a deadline for a given resource specification. Then, the WMS automatically performs resource acquisition, task scheduling, and workflow execution. In more detail, the *Resource Capacity Estimator* analyzes the workflow structure to determine the amount of computing resources. The *Resource Acquisition Module* then negotiates the external resource provisioning systems and acquires the determined amount of resources that satisfy all requirements. Once appropriate resources are allocated, the *Execution Manager* automatically runs the tasks on the resources guided by the *Scheduling Module*. Workflow tasks communicate with each other via the *storage* where all input and output data persist.

The main difference of this computing platform model from the traditional high performance computing models is that resource allocation *is application-driven*, and that resource set size can be changed at runtime.

In this computing platform model, the PBTS algorithm plays a dual role; one is to determine the resource capacity as a Resource Capacity Estimator, and the other is to determine the schedule of tasks for each charge time as a Scheduling Module. In contrast to BTS that estimates the resource capacity for the entire workflow execution before running the workflow, PBTS determines the resource capacity at the boundary of every charge time and adjusts it, if necessary, considering the status of resources and tasks.

2.2. Application model

PBTS uses the same application model with that of BTS described in [16]. An application is expressed as a workflow composed of ordered tasks building a directed acyclic graph $w = (T, E)$ where T is a set of vertexes representing n tasks $t_i \in T$, $1 \leq i \leq n$, and E denotes a set of directed edges between two tasks. An edge $e(t_i, t_j) \in E$, $t_i, t_j \in T$, $t_i \neq t_j$ implies that t_i must be executed before t_j. $P(t) \subset T$ and $C(t) \subset T$ represent the sets of immediate predecessors and successors of a task $t \in T$, respectively. Similarly, $A(t) \subset T$ and $D(t) \subset T$ denote the sets of all ancestor tasks and descendant tasks of $t \in T$, respectively. Without loss of generality, we can say that there is only one entry task t_{top} and one exit task t_{bottom}. If there are multiple entry tasks or exit tasks, we can merge them by introducing zero-weighted tasks, t_{top} spawning all entry tasks and t_{bottom} joining all exit tasks.

We also assume that each task is executed non-preemptively and exclusively on a single resource or a set of resources. Every task $t \in T$ is associated with two values given by $ET(t) \in \mathbb{N}$ and $HR(t) \in \mathbb{N}$, which represent the execution time and the host requirement of t, respectively. Typically, the execution of a task consists of three phases, downloading of input data from the storage system, running the task, and transferring output data to the storage system. Thus, $ET(t)$ is determined by the computation time of tasks and the size of input/output data. We assume that the execution time of individual tasks are given by users [19].

The host requirement, $HR(t)$, denotes the number of computing hosts that a task simultaneously occupies throughout its execution. For example, a parallel processing task such as an MPI-task requires multiple fully-connected computing hosts at the same time while a sequential task can be executed on a single host independently.

The host requirement for a task can be fixed or flexible throughout the execution. If a task needs a fixed number of resources, we call the task *rigid* while the task is *malleable* if the host requirement is changeable. For instance, parameter study applications are malleable because the degree of parallelism can be adjusted dynamically, while a sequential task or an MPI-task is rigid because the numbers of resources are statically determined by the task characteristics. In this paper, we term a malleable task as an *M-task* while a rigid task as an *R-task*, and two symbols, T_M and T_R, represent the set of *M*-tasks and *R*-tasks, respectively.

We use the simplified equation to calculate the execution time of an *M*-task as in Eq. (1) where $FET(t)$ is the execution time if the task t is executed on a single resource and $SET(t)$ is the execution time of the rigid computation part in $FET(t)$ as modeled in previous studies [20–22]. $FET(t)$ and $SET(t)$ should be given by users for every *M*-task. For *R*-tasks, $FET(t)$ and $SET(t)$ are equal to $ET(t)$.

$$ET(t) = SET(t) + \frac{FET(t) - SET(t)}{HR(t)}. \quad (1)$$

All tasks are assumed to communicate via storage instead of directly transferring of data between tasks. This model allows us to have data persistency and enables asynchronous computation. As a result, multiple instances of the same workflow and other workflows in the same domain can reuse the stored data and avoid the redundant computation. In addition, the persistent data can be used for recovery in case of application failures. On the other hand, the asynchronous computation can increase the resource utilization. The synchronous communication between tasks forces the sender to be alive until all child tasks receive the data. With asynchronous computation, the sender tasks can release their resource immediately once data persists in the storage system successfully. One limitation of this model is that the storage system or network can be a bottleneck.

2.3. Resource model

In our architecture, the Resource Provisioning System mediates between individual resource providers and applications; it acquires a collection of computing hosts from resource providers on behalf of applications, based on the published resource specifications and users' resource requirements. A *computing host* represents an independent processing unit equipped with a CPU, memory, local storage, and network interface on which any tasks of the workflow can be executed. All computing hosts allocated by the provisioning system should satisfy all requirements such as performance, availability, bandwidth, and so on, and they are completely connected to each other by a network. These computing hosts are computing resources that have similar or comparable configurations and performance per application basis. Therefore, our algorithm treats these computing hosts as homogeneous resources even though they can be actually heterogeneous because they are allocated from independent resource providers such as Grid and Compute Cloud as long as the SLA (Service Level Agreement) is satisfied. This approach makes the PBTS algorithm simple and lessens the scheduling overhead of the WMS.

We use the term, *Resource Capacity (RC)*, to represent the number of computing hosts allocated to an application workflow. The WMS can adjust the resource capacity dynamically. Since the financial cost (or simply *cost*) is calculated as the product of resource capacity and the allocation duration, we can reduce the cost by allocating the resource on-demand basis. That is, we can acquire resources just before more resources are needed and release them immediately when resources are idle. However, even though current provisioning systems support such elasticity, the cost model is restricted. For example, Amazon EC2 charges for resource usage on a per hour basis. If somebody uses a virtual machine from EC2 for any few minutes, he/she should pay for the whole hour. Therefore, in order to minimize the cost, resources should be held for the hour unit length and the resources should be utilized as densely as possible.

Our algorithm is fully aware of such a pricing policy in reality as well. Any adjustment of the resource set can take place only at the border of charge time. In other words, the application's running time is divided into time-partitions whose length is equal to the time charge unit of resource provisioning system (e.g., one hour), and the resource capacity is adjusted at each time-partition while the resource capacity remains constant within a time-partition. In this model, the cost is determined by the sum of the resource capacity over all time-partitions.

3. Related work

The issue presented in this paper is different from the well-known workflow scheduling problems. The main objective of conventional workflow scheduling algorithms is to minimize the makespan of workflow for a given resource set [23–25]. Most algorithms rely on a list scheduling and their performance is known fairly well, having relatively small time complexity [26–29]. By contrast, the goal of our study is to find the minimum set of resources for application workflow that meet the given deadline. More details about how the resource capacity estimation is different from the conventional workflow scheduling were presented in our previous study [16].

Based on this understanding, we only cover the prior studies relevant to this study and categorize them into two groups: (i) techniques that enable large-scale application execution with dynamic resource provisioning, and (ii) techniques that estimate the resource capacity for workflow execution.

As the IaaS cloud service has become mature, large-scale scientific applications are able to leverage the dynamically provisioned resources instead of using a dedicated supercomputer. For example, the Pegasus WMS [7] can manage the execution of complex application workflows on virtual machines acquired from Amazon EC2 [30,31]. As another example, Science Cloud [32] provides scientists, who have no dedicated computing resources, with the IaaS cloud service, using the Nimbus toolkit [33]. Even though its application is limited to a specific domain, Amazon's Elastic MapReduce [34] technique also enables users to execute applications over the Hadoop framework on virtual machines. However, these approaches are not able to exploit the elasticity of resources because the resource capacity, i.e., the number of virtual machines, is determined before applications start and is fixed even though resource providers allow dynamic resource management.

In the meantime, some approaches focus on adjusting the resource capacity according to the amount of demand. Murphy et al. [35] suggested a system that dynamically provisions virtual clusters by monitoring a job queue and spawns VMs to process

Condor jobs. Marshall et al. [36] introduced *Elastic Site*, a virtual cluster composed of elastically provisioned resources. Elastic Site determines when to boot up additional VMs in the cloud or to terminate them, based on the information provided by the queue sensor in order to handle the bursting of demand and avoid the low utilization at idle time. Please note that both approaches only consider the current status of the global queue but do not exploit the information of future demands even when the prediction is possible as in workflow cases.

Unlike the aforementioned approaches, we focus on minimizing the cost from the perspective of a user who wants to finish his/her workflow within a deadline. On another note, our approach enables to find the best-effort resource allocation plan, based on the workflow profile such as structure, input date size, and approximate execution time.

The PBTS algorithm attempts to estimate the resource capacity which minimizes the cost to complete a workflow within a given deadline. As a prior research, we proposed the BTS algorithm as a solution of the resource capacity estimation problem under the assumption that the resource capacity is static throughout the entire execution of the workflow [15,16]. Subsequently, we propose the PBTS algorithm that supports the elastic resource allocation/release environment and estimates the changes of resource capacities over time.

To the best of our knowledge, no existing research solves exactly the same problem as the PBTS algorithm. However, several researches aim at a goal similar to the BTS algorithm. Sudarsanam et al. [37] proposed a simple technique for estimating the amount of resources. It iteratively calculates makespans and utilizations for numerous resource configurations and finds the best one. Wieczorek et al. proposed a general mechanism for a bi-criteria workflow scheduling heuristic based on dynamic programming called DCA [38]. It generates and checks candidate schedules iteratively and finds the best among them. The problem of BTS can be solved by DCA when two criteria are the workflow's makespan and the resource capacity. Even though these approaches are likely to find an optimal solution, it does not scale well with large workflows and large resource sets.

Huang et al. [39] proposed a mechanism for finding the minimum resource collection (RC) size required to complete a workflow within the minimum execution time. The RC size is determined according to empirical data gathered from many sample workflows, by varying such parameters as the DAG size, the communication–computation ratio, parallelism, and regularity. Even though this approach achieves reasonable performance for workflows with similar characteristics to those of the sample workflows, it does not guarantee that its estimates are correct for arbitrary workflows. Additionally, the parallelism and regularity cannot be calculated deterministically for workflows with complex structures. Due to such limitations, this approach is only useful for some limited classes of workflows. By contrast, our algorithm can be applied to any type of workflow, and it does not require a kind of training phase since our algorithm directly analyzes the workflow structure. Finally, our algorithm can arbitrarily explore any desired finish times greater than the minimum execution time.

4. The PBTS algorithm

4.1. Problem specification

The goal of this study is to determine a resource provisioning plan of workflow; the algorithm should determine the number of computing hosts and the task schedule for each time-partition. Based on the models discussed in Section 2, PBTS first captures the application characteristics and the resource constraints by three inputs: (1) a workflow profile represented by an acyclic graph $W = (T, E)$ with $ET(t)$ and $HR(t)$ of all rigid tasks ($t \in T_R$), and $FET(t)$ and $SET(t)$ of all malleable tasks ($t \in T_M$), (2) application deadline (X_D), and (3) length of time-partition (X_P), which is equal to the charge time unit of resources. A schedule of task t includes its scheduled start time $ST(t)$, its host requirement $HR(t)$, and its adjusted execution time $ET(t)$. The schedule must satisfy the conditions presented in Eq. (2) to guarantee the precedence between tasks and the application deadline.

$$\begin{aligned} &\forall e(t_i, t_j) \in E, \quad ST(t_i) + ET(t_i) \leq ST(t_j) \\ &\forall t \in T, \quad ST(t) + ET(t) < X_D. \end{aligned} \tag{2}$$

The resource capacity of time-partition is dependent on the number of tasks scheduled at the time-partition. In Eq. (3), $NH(S, x)$ denotes the number of computing hosts allocated for a set of tasks (S) at time x. The resource capacity of time-partition ($RC(i)$) then is the peak number of computing hosts within the time-partition (Eq. (4)).

$$NH(S, x) = \sum_{\{t \mid ST(t) \leq x < ST(t)+ET(t), t \in S\}} HR(t) \tag{3}$$

$$RC(i) = \max_{(i-1)\times X_P \leq x < i \times X_P} NH(T, x), \quad \text{where } 0 < i \leq n_p. \tag{4}$$

The objective of PBTS is to minimize the total cost ($Cost_{total}$) by finding the best task schedule for each time-partition. Eq. (5) defines the total cost, where $Cost_{unit}$ is the price when a single computing host is used for one time charge unit, i.e., X_P. The total cost is proportional to the sum of resource capacity across all time-partitions.

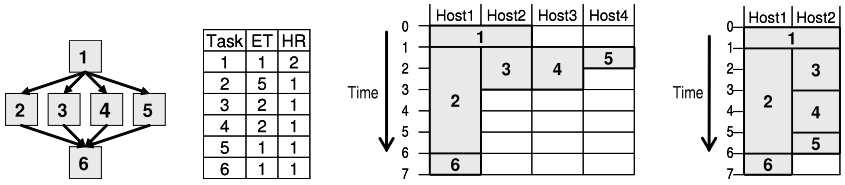
$$Cost_{total} = Cost_{unit} \cdot \sum RC(i). \tag{5}$$

4.2. Approach

As emphasized earlier, our goal of this study is to minimize the resource cost, not the makespan of workflow. PBTS is based on the key technique that a task can be delayed as long as it does not break time constraints of the application, anticipating that other tasks can exploit the artificial slack time. If the deadline (X_D) is longer than the length of the critical path of workflow, any tasks can be possibly delayed. Even when the deadline is equal to the minimum makespan of workflow, tasks on non-critical paths can be delayed as well.

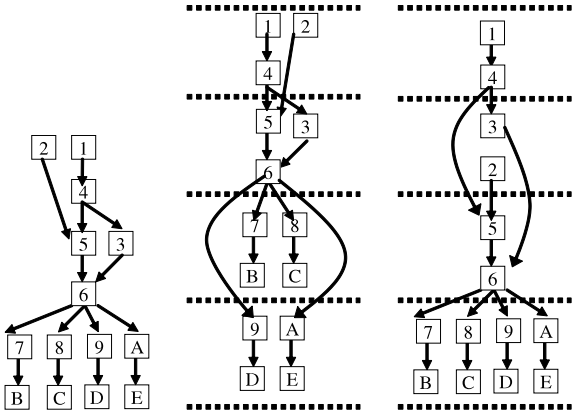
Fig. 2 illustrates how this simple idea can reduce the number of resources. For comparison, we present a naive approach that schedules all tasks as early as possible. Fig. 2(a) shows the structure and task properties ($ET(t)$ and $HR(t)$) of the workflow. The naive approach estimates that four processors are required to exploit the maximum parallelism of the workflow as in Fig. 2(b). With four computing hosts, this workflow spends seven time units, which is the sum of execution times of tasks on the longest path (i.e., tasks 1, 2, and 6). However, tasks 4 and 5 do not utilize the resources all the time. We can delay tasks 3, 4, and 5 arbitrarily without affecting the makespan of the workflow. For instance, we can execute tasks 3, 4, and 5 sequentially on a single host, because the sum of their execution times is equal to the execution time of task 2. Fig. 2(c) shows that we can schedule all tasks with the same makespan by using only two hosts.

PBTS is an extension of the BTS algorithm for elastic resources. PBTS and BTS are identical if a workflow runs within a single time-partition, i.e., $X_D \leq X_P$. However, if a workflow runs over multiple time-partitions, PBTS becomes an online algorithm, and it leverages BTS iteratively to estimate the resource capacity for individual time-partitions. That is, PBTS adaptively estimates the resource capacity of time-partition at runtime while the decision of BTS is made at the beginning of execution and fixed throughout the



(a) Workflow structure. (b) Task information. (c) Naive scheduling. (d) Optimal scheduling.

Fig. 2. Comparison of two scheduling strategies for minimum makespan of an example workflow.



(a) Workflow structure. (b) BTS approach. (c) PBTS approach.

Fig. 3. An example scenario that the elasticity reduces the cost.

execution. Fig. 3 shows the difference with an example workflow having 14 identical tasks ($ET(t) = 1$). The minimum makespan of the workflow is six when $X_D = 8$ and $X_P = 2$. BTS estimates the resource capacity as two and keeps them throughout. By contrast, PBTS decides to use only one computing host for the first three time-partitions and expands the resource capacity for the last time-partition. As a result, PBTS can execute the same workflow within the same deadline only with the 7/8 cost.

A simple approach people may try to solve the problem is to apply BTS repeatedly for all unexecuted tasks at every time-partition. That is, given a set of unfinished tasks and a deadline of charge time unit, BTS can estimate the minimum resource capacity for each time-partition until all tasks finish. However, this approach can fall into a local optimum because it does not consider the consequence of a decision for each partition on the total cost. In contrast, PBTS takes into account of the influence on the total cost when estimating the minimum resource capacity of each partition.

The high-level description of PBTS is given in Fig. 4 where T_{remain}, T_{sche}, and T_{exec} denote the set of unexecuted tasks, the set of tasks selected to be scheduled at the target time-partition, and the set of executed tasks, respectively. PBTS iterates all time-partitions of the application one by one in the time order at runtime, and has three phases to determine the resource capacity of the next time-partition; (1) PBTS identifies the set of executable tasks for the next partition, based on the approximate resource capacity, considering the total cost, (2) estimates the exact resource capacity and the schedule of the selected tasks for the time-partition using BTS, (3) allocates actual resources as estimated and executes the selected tasks on the resources by the schedule. PBTS performs the estimate for the first time-partition before the workflow starts, considering all tasks in the workflow. PBTS then repeats the three phases for the remaining time-partitions until all tasks are completed. PBTS considers the actual progress information of tasks in the current partition, and conducts the estimate for the upcoming partition just before the next partition starts.

This online approach has two main advantages. First of all, since the algorithm monitors the task progress, the algorithm can adapt to the dynamics of tasks and resources in each partition, and can adjust the resource capacity in the following partitions. For example, if tasks are lagged or unexecuted in the current partition, PBTS continues to execute such tasks, and tries to allocate additional resources for them in the next partition to keep the deadline. Likewise, if tasks finish earlier, PBTS aggressively utilizes the free resources by executing the unscheduled tasks in advance. As a result, PBTS can estimate less resource capacity in the following time-partitions, which eventually can reduce the total cost. Second, since the workflow scheduling overhead is amortized over the iterations, applications experience only the scheduling overhead for the first time-partition even with large workflows. Note that this online approach is only feasible with a highly efficient and scalable algorithm like PBTS. The following subsection describes each phase of the iteration in detail.

4.3. Task selection phase

This phase is a major difference from our previous study where the algorithm, BTS, which assumes a single virtual time-partition equal to the application deadline regardless of actual time-partitions determined by resource providers and determines the time schedule of all tasks at once. To the contrary, PBTS is aware of the time partitions and experiences the dynamic changes of tasks set and resource capacity in every time-partition. PBTS should determine the distribution of tasks across time-partitions in order to determine the resource capacity of each time-partition. The task selection phase identifies the task set to be scheduled in the immediate time-partitions. However, the task set is also influenced by the resource capacity of the time-partition, and this constitutes a cyclic dependency between resource capacity estimate and task set selection. To resolve this issue, the task

1: $T_{remain} \leftarrow T$, $T_{exec} \leftarrow \emptyset$
2: **FOR** i increased from 1 to n_p **DO**
3: **Task selection phase:**
: Select $T_{sche} \subset T_{remain}$ to be scheduled in i-th time-partition
4: **Resource capacity estimate and task scheduling phase:**
: Apply BTS on T_{sche} to decide $RC(i)$ and $ST(t), \forall t \in T_{sche}$
5: **Task execution phase:**
: Acquire $RC(i)$ computing hosts
: Execute all tasks in T_{sche}, add executed tasks into T_{exec}
6: $T_{remain} \leftarrow T_{remain} - T_{exec}$
7: **END FOR**

Fig. 4. The high-level description of the PBTS algorithm.

selection phase first approximates the resource capacity of all remaining time-partitions with the given information of the workflow. Then, PBTS determines a set of candidate tasks that can be scheduled on the immediate time-partition under the assumption that the approximation is accurate. Note that the task selection phase does not try to minimize the resource capacity of the target time-partition greedily. Rather, it determines the resource capacity of the time-partition, considering the total cost of the entire workflow and the deadline in order to avoid falling into a local optimum.

The workflow structure and the deadline (X_D) determine the resource capacity of each time-partition. Especially, the degree of parallelism of tasks is a dominant factor of resource capacity when the deadline is close to the minimum makespan. However, if the deadline is far from the minimum makespan, tasks can be delayed as long as the deadline is guaranteed. In this paper, the difference between the deadline and the minimum makespan is called the *spare time* (X_{spare}). PBTS exploits this spare time to reduce resource capacity. The distribution scheme of the spare time affects the overall cost. For example, the workflow in Fig. 5(a) needs five unit computing hosts to finish by the minimum makespan (i.e., two time-partitions): three for the first time-partition and two for the second time-partition. Let X_D be three times of X_P and have the spare time of one X_P. If the spare time is used for the first four tasks as in Fig. 5(b), one computing host allocated for two time-partitions can execute them, and two computing hosts allocated for one time-partition can execute the remaining four tasks. The total cost of this execution is 4 units. On the contrary, if the spare time is assigned to the last four tasks as in Fig. 5(c), the first four tasks still need three computing hosts, and one computing host is enough for the last four tasks. The cost of this execution is 5 units. As such, finding the best distribution of the spare time is essential to approximating the resource capacity with minimal cost.

In summary, the task selection phase consists of three sub-steps: (1) analyzing the workflow structure to get the variance of resource capacity over time to finish the workflow as early as possible, (2) finding the best distribution of the spare time that minimizes the total cost and approximating the resource capacity of the target time-partition, (3) selecting the set of tasks to be scheduled in the target time-partition for the resource capacity approximated in Step (2).

4.3.1. Workflow structure analysis

The first step of the task selection phase is to identify the distribution of resources over time-partitions. The resource capacity distribution is closely related to the structure and characteristics of workflow. For this purpose, we quantify the workflow structure and characteristics via a simple mathematical model.

We analyze the structure of the workflow by using the earliest start time and the latest finish time of un-scheduled tasks under the assumption that the deadline (X_D) is equal to the minimum makespan of workflow (X_M). As shown in Eq. (6), the earliest start time of a task, i.e., $EST(t)$, is influenced by the start time of scheduled parent tasks and the earliest start time of unscheduled parent tasks. We use symbols, x_{ps} and x_{pe}, to represent the start time and the end time of the target time-partition, respectively. The minimum makespan of the workflow is equal to the earliest start time of the zero-weighted bottom task, i.e., $X_M = EST(t_{bottom})$. $LFT_{\min}(t)$ is the latest finish time of the task when $X_D = X_M$(Eq. (7)). A task can be scheduled anytime between $EST(t)$ and $LFT_{\min}(t)$ as long as the workflow makespan remains smaller than or equal to X_M. We can calculate these values by a depth first search from t_{top} and t_{bottom}.

$$EST(t) = \max_{\substack{\forall j \in P(t) - T_{exec}, \forall k \in P(t) \cap T_{exec} \\ ST(k) + ET(k), x_{ps}\}}} \{EST(j) + ET(j), \tag{6}$$

$$LFT_{\min}(t) = \min_{\forall c \in C(t)} \{LFT_{\min}(c) - ET(c), X_M\}. \tag{7}$$

As a part of the process determining $EST(t)$ and $LFT_{\min}(t)$, PBTS also initializes ET and HR of M-tasks. As shown in Fig. 6, the goal of this initialization algorithm is to extend ETs of M-tasks not on the critical path as long as possible and to reduce their HRs. As a result, the host requirements of all M-tasks are balanced.

PBTS quantifies the workflow structure by $AHR(x)$ which denotes the aggregate host requirements of all tasks that can be scheduled at time x measured from x_{ps}. All tasks on the critical path must be scheduled as early as possible for the minimum makespan. However, the schedulable range of tasks on the non-critical paths is wide, and it is hard to decide at which time non-critical tasks should be scheduled in order to determine $AHR(x)$. For simplicity, we use the average of two extreme scheduling schemes as shown in Eq. (8).

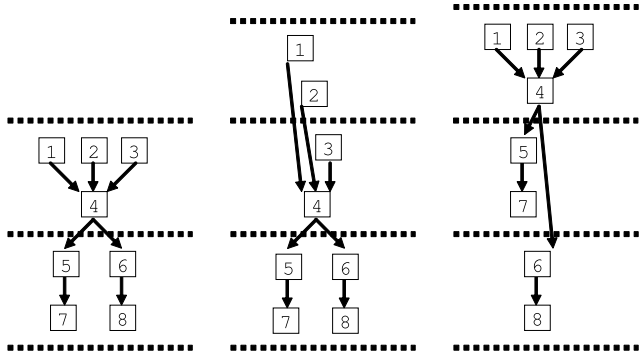
$$AHR(x) = \frac{AHR_e(x) + AHR_l(x)}{2} + NH(T_{exec}, x). \tag{8}$$

$AHR_e(x)$ defines the sum of the average host requirement of all tasks whose schedulable range is intersected on x when all tasks are scheduled as early as possible (cf. Eq. (9)). $LFT_e(t)$ is the latest finish time restricted by its child tasks as in Eq. (10).

$$AHR_e(x) = \sum_{\forall t \in \{t | EST(t) \le x < LFT_e(t)\}} \frac{ET(t) \cdot HR(t)}{LFT_e(t) - EST(t)} \tag{9}$$

$$LFT_e(t) = \min_{\forall t_c \in C(t)} EST(t_c). \tag{10}$$

On the contrary, $AHR_l(e)$ defined in Eq. (11) is the sum of average host requirements of all tasks whose schedulable range is intersected with the minimum makespan ($X_D = X_M$). In $AHR_l(e)$, the schedulable range of tasks is from $EST_l(t)$ to $LFT_{\min}$ where



(a) Two time-partitions. (b) Three time-partitions, two for the first four tasks. (c) Three time-partitions, two for the last four tasks.

Fig. 5. The cost with different spare time and its distribution.

1: **FORALL** $t \in T_{remain}$ **DO** $ET(t) \leftarrow SET(t) + 1, HR(t) \leftarrow FET(t) - SET(t)$
2: Calculate $EST(t)$ and $LFT_{min}(t)$, $t \in T_{remain}$ through depth first searches.
3: $T_{red} \leftarrow T_{remain} \cap \{$ t is not on the critical path and $HR(t)$ is greater than 1$\}$
4: **LOOP WHILE** $T_{red} \neq \emptyset$ **DO**
5: $t_t \leftarrow t \in T_{red}$ with the largest HR
6: $ET(t_t) \leftarrow ET(t_t) + 1$, $HR(t_t) \leftarrow \lceil \frac{FET(t_t) - SET(t_t)}{ET(t_t) - SET(t_t)} \rceil$
7: Update $EST(t_j)$ of all $t_j \in D(t_t)$ and $LFT(t_k)$ of all $t_k \in A(t_t)$
8: $T_{red} \leftarrow T_{red} - \{tisonthecriticalpath \vee HR(t) = 1\}$
9: **END LOOP**

Fig. 6. Initializing HR and ET of every M-task.

$EST_l(t)$ in Eq. (12) is the earliest start time when all its parent tasks are finished at their $LFT_{\min}$.

$$AHR_l(x) = \sum_{\forall t \in \{t | EST_l(t) \le x < LFT_{\min}(t)\}} \frac{ET(t) \cdot HR(t)}{LFT_{\min}(t) - EST_l(t)} \tag{11}$$

$$EST_l(t) = \max_{\forall t_p \in P(t)} LFT_{\min}(t_p). \tag{12}$$

Finally, the last term of Eq. (8) represents the number of computing hosts occupied by tasks that started in the previous time-partitions but have not finished yet. Tasks having longer execution times than X_P or scheduled near the end of the previous time-partition can run over multiple time-partitions.

4.3.2. Spare time distribution

Based on the information of resource requirement over time obtained in the previous step, this step determines which time-partition can have the most benefit with the spare time. The resource capacity of the time-partition is determined by the peak resource requirement within the time-partition. Therefore, the beneficiary of spare time should be able to lower the peak at the expense of the task execution time.

As illustrated in Fig. 7(a), $AHR(x)$ of workflow can be represented by a plane where the x-axis represents time and the y-axis, the resource capacity. As shown in Fig. 7(b), the plane then can be partitioned along with the time axis, based on the charge time unit. In this plane, the total resource requirement of workflow is equal to the total area of the polygon. Since the resource capacity changes over time, the polygon can have an arbitrary stepped shape. However, we approximate the shape for each partition to an L-shaped hexagon with the same area to the original polygon in order to simplify the area computation.

Fig. 8 illustrates this transforming process. The height of the leftmost side of hexagon, $H_i^{\max}$, is equal to the tallest height of the original polygon while the height of the rightmost side, $H_i^{\min}$, is equal to the shortest height of the original polygon. Then, the width of the left top side can be calculated by Eq. (13).

$$x_i^{\max} = \begin{cases} X_P & \text{if } H_i^{\max} = H_i^{\min} \\ \dfrac{S_i - X_P \cdot H_i^{\min}}{H_i^{\max} - H_i^{\min}} & \text{if } H_i^{\max} > H_i^{\min} \end{cases}$$

$$\text{where } S_i = \sum_{x=(i-1)\cdot X_P}^{i \cdot X_P} AHR(x). \tag{13}$$

Now, we transform this simplified hexagon one more time when extra time is assigned to the partition. We just transform the rectangle on the left side of the hexagon while the right size remains the same, which means that tasks scheduled on the busiest time utilize the additional time to reduce the resource requirement. Then, the new height of the final hexagon can be calculated by Eq. (14), and the total cost of this partition is obtained by Eq. (15).

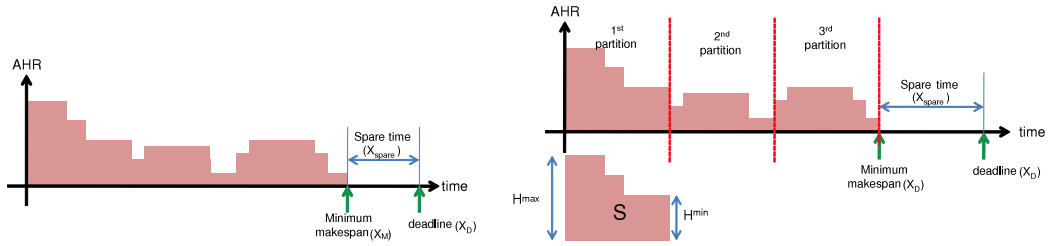
$$H_i^{new} = H_i^{\max} \cdot \frac{x_i^{\max}}{x_i^{\max} + x_{inc}} \tag{14}$$

$$Cost_i = H_i^{new}(X_P + x_{inc}). \tag{15}$$

Our interest here is which partition can have the most benefit with the spare time. To get the optimal answer, we need to solve an integer program. For simplicity, we compare a differential coefficient of the cost function (Eq. (16)). The $dCost/dx_{inc}$ value of candidate time-partition should be less than zero to reduce the total cost, and the smaller $dCost/dx_{inc}$ means more cost reduction. Therefore, the time-partition with the smallest differential coefficient is the best candidate.

$$\frac{dCost_i}{dx_{inc}} = H_i^{\max} \cdot x_i^{\max} \cdot \frac{x_i^{\max} - X_P}{(x_i^{\max} + x_{inc})^2}. \tag{16}$$

Next, we apply the new height and width to the original polygon of resource capacity, and reshape it. For instance, the polygon on



(a) A plane graph of AHR. (b) Partitioning.

Fig. 7. Plane graph expression of AHR and its partitioning.

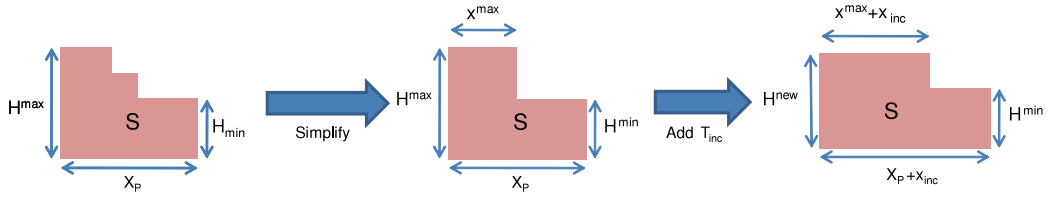


Fig. 8. Cost approximation model.

the left in Fig. 9 is transformed into the polygon on the right after trimming the tall rectangles down to H_i^{new}. If the height of the rectangle is shorter than H_i^{new}, reshaping is not required.

Let $w_{i,k}$ and $h_{i,k}$ be the width and the height of k-th rectangle, respectively, and assume that there are r_i rectangles in the i-th time-partition. The width and the height of k-th rectangles of i-th time-partition are updated by Eq. (17) and $AHR(x)$ is also updated accordingly.

$$\begin{aligned} h_{i,k} &\leftarrow \min\{h_{i,k}, H_i^{new}\} \\ w_{j,k} &\leftarrow h_{i,k} \cdot w_{j,k} / \min\{h_{i,k}, H_i^{new}\}. \end{aligned} \tag{17}$$

PBTS repeats this distribution until the entire spare time is consumed, or all time-partition $dCost/dx_{inc}$ values become zero meaning that each L-shaped hexagon becomes a rectangle, i.e., $H^{\max} = H^{\min}$. The termination conditions make tasks scheduled as early as possible if adding the spare time to any time-partition cannot reduce the cost. As such, PBTS minimizes the turnaround time of workflow as well as the cost.

The time unit of the spare time distribution, x_{inc}, should be determined to balance the computing overhead and the accuracy of approximation. According to our experiments, the rule of thumb is to use one tenth of X_P. We will use a symbol x_{inc}^{unit} to represent the amount of spare time assigned to a time-partition.

4.3.3. Task selection

As the last step of the task selection phase, PBTS selects tasks to be scheduled in the target time-partition by checking the *LFT* of tasks extended through spare time distribution.

In $AHR(x)$ graph, X-axis of $AHR(x)$ graph represents the time from x_{ps} and the rightside edge of a rectangle represents $LFT_{\min}(t)$ of a task. Thus, $LFT_{\min}(t)$ can be calculated as the sum of width of all rectangles located at the left-side of the matched rectangle. Assigning a portion of spare time to a time-partition extends the schedulable range of tasks to be scheduled on the time-partition. More formally, as in Fig. 9, the widths of some rectangle ($w_{i,k}$) and correspondingly the $LFT_{\min}(t)$s of some tasks are increased. We use a symbol $LFT^l_{\min}(t)$ to represent the increased value of $LFT_{\min}(t)$ of task t, and it is calculated as in Eq. (18) when t is matched to k-th rectangle in p-th polygon.

$$LFT^l_{\min}(t) = x_{ps} + \sum_{n=1}^{p-1} \sum_{m=1}^{r_n} w_{n,m} + \sum_{m=1}^{k} w_{p,m}. \tag{18}$$

In the task selection phase, first of all, PBTS sets T_{sched} to an empty set. Then for every task t in T_{remain}, if $LFT^l_{\min}(t) - ET(t)$ is smaller than x_{pe}, task t is added to T_{sche} since t is considered that its scheduling range is laid on the target time-partition. After checking all tasks, the tasks selection phase returns T_{sched}.

All the steps of the task selection phase which select the tasks to be executed in the target time-partition among unscheduled tasks (T_{remain}) are summarized in Fig. 10.

4.4. Resource capacity estimate and task scheduling phase

PBTS applies the BTS algorithm to the tasks selected in the previous phase to estimate the minimum resource capacity of the adjusted time-partition. More specifically, the task placement phase and the task redistribution phase of the BTS algorithm find the best time schedules of tasks [16]. The only difference is that $LFT(t)$ of task is bounded by the end of time-partition, x_{pe}, as in Eq. (19). Note that tasks may not complete their computations within the time-partition, but all tasks must start before the end of the time-partition.

$$LFT(t_i) = \min_{\forall t_j \in C(t_i) \cap T_s} \{LFT(t_j) - ET(t_j), LFT^l_{\min}(t)\}. \tag{19}$$

First, BTS performs preliminary scheduling of tasks, using a heuristic algorithm as presented in Fig. 11, where T_s represents the set of tasks whose start time is set by BTS. This algorithm iterates three steps until the start times of all tasks are determined. BTS first selects a task with the smallest schedulable duration among unscheduled tasks (line 2) and then determines the best start time of the task within its schedulable duration (lines 3, 4). Finally, it updates *EST*s and *LFT*s of all its dependent tasks to reflect the changes in the time constraints of the task (lines 5, 6). Once the task placement phase finishes, the start time ($ST(t)$) of every task is determined.

Since the task placement algorithm does not always find an optimal solution, BTS does further optimization in the task redistribution phase as in Fig. 12. For each task placed in the most resource-demanding time slots ($t_b \in T_{busy}$), BTS tries to relocate them between $EST_{\min}(t_b)$ and $LFT_{\max}(t_b)$ or to change its host requirement ($HR(t_b)$) in order to reduce the maximum $NH(T, x)$ of the time slot on which the task is placed. In other words, tasks in busy time slots are moved to relatively idle time slots and eventually the total resource requirement over time is balanced. A relocation can involve cascading relocations of dependent tasks, in order to preserve the precedence between tasks. Whenever the new start time of relocated task is earlier

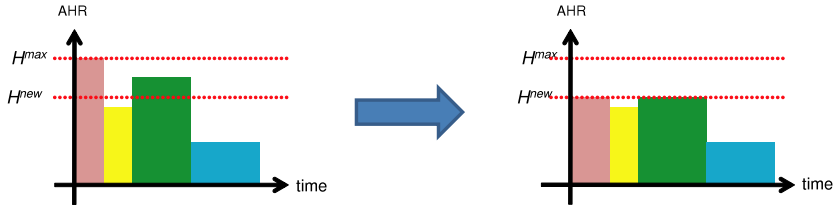


Fig. 9. Change of AHR of the most benefitable time-partition after X_{inc} is added.

TaskSelection(T_{remain})

1: Build the $AHR(x)$
2: Calculate the spare time, $X_{spare} \leftarrow X_D - EST(t_{bottom})$
3: **WHILE** $X_{spare} > 0$ **DO**
4: partition $AHR(x)$ diagram into m polygons whose width are X_P
5: Find the time-partition with smallest $\frac{dCost_i}{dX_{inc}}$
6: **If** there is no time-partition which satisfies $\frac{dCost_i}{dX_{inc}} < 0$, **BREAK WHILE**
7: Add x_{inc}^{unit} to the selected time-partition.
8: update $w_{i,k}$, $h_{i,k}$, and $AAO(x)$
9: $X_{spare} \leftarrow X_{spare} - x_{inc}^{unit}$
10: **END WHILE**
11: $T_{sche} \leftarrow \{t \mid t \in T_{remain}, LFT_{min}^{l}(t) - ET(t) < x_{pe}\}$
12: **RETURN** T_{sche}

Fig. 10. Algorithm description of the task selection phase.

ALGORITHM Task placement

1: **LOOP WHILE** $T_s \neq T$ **DO**
2: $t_t \leftarrow t \in T - T_s$ with the narrowest $SD(t)$,
: if tied, the largest $HR(t)$, and the largest $n(T - A(t) \cup D(t))$
3: Let $peak(t, st) = \max_{x=st}^{st+ET(t)} NH(T_s, x), minPeak \leftarrow \min_{st=EST(t_t)}^{LFT(t_t)-ET(t_t)} peak(t_t, st)$,
: $STC \leftarrow \{st \mid peak(t_t, st) = minPeak, EST(t_t) \leq st \leq LFT(t_t) - ET(t_t)\}$
4: **IF** $\frac{\sum_{\forall t_a \in A(t_t)} ET(t_a)HR(t_a)}{EST(t_t)+|SD(t_t)|} \leq \frac{\sum_{\forall X_D \in D(t_t)} ET(X_D)HR(X_D)}{RFT-LFT(t_t)+|SD(t_t)|}$ **DO**
: $ST(t_T) \leftarrow$ the earliest $st \in STC$
: **ELSE DO** $ST(t_T) \leftarrow$ the latest $st \in STC$ **END IF**
5: $T_s \leftarrow T_s \cup \{t_t\}$ and recalculate $NH(T_s, x)$
6: Update $EST(t_j)$ of all $t_j \in D(t_t)$ and $LFT(t_k)$ of all $t_k \in A(t_t)$
7: **END LOOP**

Fig. 11. Task placement algorithm of BTS.

than the finish time of any precedent tasks, the preceding task must be relocated also before the new start time. The new start time is selected as closely as possible to the original, to minimize the impact on dependent tasks. BTS first tries to relocate tasks to earlier time slots in ascending order of $ST(t)$ until no relocation can reduce the maximum $NH(T, x)$ by *MoveLeft* (line 2–4). BTS then tries to relocate tasks to later time slots in descending order of $ST(t) + ET(t)$ by *MoveRight* (line 5–7). BTS stops when no more relocation is possible, and returns the maximum value of the resulting $NH(T, x)$ as the resource capacity for the workflow (line 8).

Both phases have a polynomial complexity. Details of the algorithms are described in our previous paper [16].

4.5. Task execution phase

With the PBTS algorithm, an estimate of resource capacity assumes a certain schedule of tasks for each time-partition. A workflow can meet the deadline with the estimated resources when the PBTS algorithm is used to schedule tasks while other algorithms may not guarantee the deadline with the PBTS's estimate. For this reason, we enforce the PBTS schedule in the execution phase.

PBTS can execute a task whenever the following three rules are satisfied. First, all parent tasks must be finished. Second, $HR(t)$ of a task must be smaller than the number of free computing hosts. Last, the execution of a task must not delay any tasks whose scheduled start time is earlier than $ST(t)$. In other words, a task can be executed only if the inequality in Eq. (20) is satisfied.

$$\sum_{j \in \{j \mid ST(t_j) < ST(t), t_j \in T_{sche}\}} HR(t_j) + HR(t) > RC_{free}. \tag{20}$$

Even though PBTS estimates the resource capacity of a time-partition very tightly, a time-partition can have free resources even when all tasks scheduled for the time-partition are executed. This is because a provisioning system can allocate over-qualified resources or tasks can run shorter than the predicted execution time.

The task execution phase tries to run unscheduled tasks to utilize the free resources. This can reduce the resource capacity

ALGORITHM Task redistribution

1: $maxNH \leftarrow \max_{x=0}^{RFT-1} NH(T,x)$, $T_{busy} \leftarrow \{t \mid \max_{x=ST(t)}^{ST(t)+ET(t)-1} NH(T,x) = maxNH\}$
2: **FOR** $t_b \in T_{busy}$ in ascending order of $ST(t)$ **DO**
3: **IF** $MoveLeft(t_b, ST(t_b)+ET(t_b))$ returns $true$ **DO**
: update $maxNH$ and T_{busy}, and restart from line 2 **END IF**
4: **END FOR**
5: **FOR** $t_b \in T_{busy}$ in descending order of $ST(t)+ET(t)$ **DO**
6: **IF** $MoveRight(t_b, ST(t_b)))$ returns $true$ **DO**
: update $maxNH$ and T_{busy}, and restart from line 6 **END IF**
7: **END FOR**
8: **RETURN** $RC \leftarrow \max_{x=0}^{RFT-1}\{NH(T,x)\}$

boolean MoveLeft(task t**,time** $bound$**)**

1: **FOR** st decreases from $bound - ET(t)$ to $EST_{min}(t)$ **DO**
2: **FOR** et decreases from $\min\{FET(t), bound - st\}$ to $\min\{ET(t), SET(t)+1\}$ **DO**
3: **IF** $t \in T_M$ **DO** $hr \leftarrow \lceil \frac{FET(t)-SET(t)}{et-SET(t)} \rceil$ **ELSE** $hr \leftarrow HR(t)$ **END IF**
4: **IF** $maxNH > \max_{x=st}^{st+et-1}\{NH(T-(A(t)\cup\{t\}),x)\} + hr$ **DO**
: goto line 8 **END IF**
5: **END FOR**
6: **END FOR**
7: **RETURN** $false$
8: **FOR** $t_{prop} \in \{t_p \mid t_p \in P(t), st < ST(t_p)+ET(t_p)\}$ **DO**
: Call $MoveLeft(t_{prop}, st)$ **END FOR**
9: **IF** all calls in line 8 return $true$ **DO**
10: $ST(t) \leftarrow st$, $ET(t) \leftarrow et$, $HR(t) \leftarrow hr$, and **RETURN** $true$
11: **ELSE RETURN** $false$ **END IF**

boolean MoveRight(task t**,time** $bound$**)**

1: **FOR** ft increases from $bound + ET(t)$ to $LFT_{max}(t)$ **DO**
2: **FOR** et decreases from $\min\{FET(t), ft - bound\}$ to $\min\{ET(t), SET(t)+1\}$ **DO**
3: **IF** $t \in T_M$ **DO** $hr \leftarrow \lceil \frac{FET(t)-SET(t)}{et-SET(t)} \rceil$ **ELSE** $hr \leftarrow HR(t)$ **END IF**
4: **IF** $maxNH > \max_{x=ft-et}^{ft-1}\{NH(T-(D(t)\cap\{t\}),x)\} + hr$ **DO**
: goto line 8 **END IF**
5: **END FOR**
6: **END FOR**
7: **RETURN** $false$
8: **FOR** $t_{prop} \in \{t_c \mid t_c \in C(t) \wedge ST(t_c) < ft\}$ **DO**
: Call $MoveRight(t_{prop}, ft)$ **END FOR**
9: **IF** all calls in line 8 return $true$ **DO**
10: $ST(t) \leftarrow ft - et$, $ET(t) \leftarrow et$, $HR(t) \leftarrow hr$, and **RETURN** $true$
11: **ELSE RETURN** $false$ **END IF**

Fig. 12. Task redistribution algorithm of BTS.

of upcoming time-partitions. PBTS selects the task satisfying three conditions, (1) all parent tasks of the task must be finished, (2) the number of free computing hosts is equal to or larger than the host requirement of the task, and (3) the tasks must finish before the end of the current time-partition. If multiple tasks satisfy the conditions, PBTS executes the task with the largest value of $ET(t) \cdot HR(t)$ first.

5. Evaluation

We evaluate our algorithm in terms of the cost and the quality of adaptation to application dynamics. For cost, we compare PBTS to a theoretical optimal solver and BTS. For adaptation, we simulate the actual task execution by using a probabilistic model based on the predicted execution time, and measure the total cost and the actual turnaround time of the workflow.

5.1. Evaluation methodology

First, we compare PBTS to the optimal solution. The optimum cost is the cost when resource allocation/release is fully elastic, and the charge unit is not discrete (e.g., one hour). Hence, the cost is exactly proportional to the resource time used by a workflow. This optimal cost is the lower bound of cost for estimation algorithms. In addition, we compare PBTS to our static algorithm, BTS, to demonstrate the importance of an algorithm being aware of the elasticity of resources. BTS aims to minimize the cost by statically estimating the resource capacity for workflows [15,16]. Instead, we enhance the BTS algorithm for fair comparison so that it can adjust the resource capacity for each time-partition, based on the BTS schedule. We call this enhanced version of BTS as BTSE.

In summary, the equations from (21) through (24) present the costs of four different approaches: Optimum, BTS, BTSE, and PBTS. The optimum cost ($Cost_{opt}$) is exactly proportional to the resource

time used by a workflow (Eq. (21)). The BTS cost ($Cost_{BTS}$) is the cost when the resource capacity is constant throughout the workflow execution (Eq. (22)). The BTSE cost ($Cost_{BTSE}$) is the cost when tasks are scheduled just one time by BTS but the resources are allocated elastically, based on the actual resource capacity requirement as in Eq. (23). $Cost_{BTSE}$ must be smaller than or equal to $Cost_{BTS}$ since BTSE can release idle resources for the under-utilized time-partitions. Finally, $Cost_{PBTS}$ is the aggregate resource capacity across all time-partitions (Eq. (24)).

$$Cost_{opt} = \sum_{\forall t \in T} ET(t) \cdot HR(t) \tag{21}$$

$$Cost_{BTS} = RC_{BTS} \cdot X_D \tag{22}$$

$$Cost_{BTSE} = X_P \cdot \sum_{i=1}^{n_p} \{ \max_{(i-1)\times X_P \le x < i \times X_P} NH(T, x) \} \tag{23}$$

$$Cost_{PBTS} = X_P \cdot \sum_{i=0}^{n_p} RC(i). \tag{24}$$

The unit of cost in the above equations is different from that of the charge time unit of resource allocation. For instance, the typical unit of execution time is a second while that of a charge time unit is an hour. As such, the actual financial cost is calculated by dividing the total cost by the charge time unit and then by multiplying the unit cost ($Cost_{unit}$). For example, let $Cost_{PBTS}$ be 20,000 and the unit of $ET(t)$ be a second. If the resource provider charges 0.1$ for one-hour usage of a virtual machine, the actual cost is $20{,}000 \times 0.1/3600$ dollars.

5.2. Real application workflow

We perform experiments with five real application workflows used in diverse scientific domains: Montage [1], CyberShake [4], Epigenomics [40], LIGO Inspiral Analysis Workflow [41], and SIPHT [42]. Montage creates custom image mosaics of the sky on-demand and consists of four major tasks: re-projection, background radiation modeling, rectification, and co-addition. CyberShake is used by the Southern California Earthquake Center to characterize earthquake hazards. Epigenomics is used to map the epigenetic state of human cells on a genome-wide scale. The DNA sequence data is split into several chunks, and then conversion and filtering is performed on each chunk in parallel. The final data are aggregated to make a global map. The LIGO Inspiral Analysis Workflow is used by the Laser Interferometer Gravitational Wave Observatory to detect gravitational waves in the universe. The detected events are divided into smaller blocks and checked. SIPHT automates the search for sRNA encoding-genes for all bacterial replicons in the National Center for Biotechnology Information database. SIPHT is composed of a variety of ordered individual programs on data. The structure of each workflow with the computation time and input/output data size of each task are summarized in Fig. 13. Further information on these workflows is available in [19]. Note that the computation time and the data size in Fig. 13 are representative and they can be reconfigured according to the problem size while the structure remains the same.

Bharathi et al. also provide a tool to generate DAX (Directed Acyclic Graph in XML) of five applications for a given workflow size, i.e., the number of tasks. The DAX file contains all information that PBTS requires such as the list of tasks, dependencies between tasks, computation time, and input/output data size of task.

We apply BTS and PBTS to the DAX files of five applications and calculate the costs of four approaches. The results are presented in Fig. 14. The minimum makespans of applications are several hours, and the numbers of tasks, $n = |T|$, range from 200 to 10,000. We choose the deadlines close to the minimum makespan (X_M) to reflect the reality that users typically prefer finishing as early as possible.

The length of the time-partition is set to one hour by default. However, for some cases with too short a deadline, we use 20 or 30 min to understand the effects of the length of charge time unit. As shown in the Montage, LIGO, and SIPHT cases, PBTS and BTSE can be benefited from short time-partitions when the resource capacity changes frequently over time.

The results show that PBTS is better than BTS and BTSE in most cases. Noticeably, the cost reduction by relaxing the deadline is remarkable with the Montage and SIPHT workflows whose degree of parallelism varies widely. In addition, PBTS is comparable to or slightly better than BTSE with CyberShake and Epigenomics. Since both estimate the costs close to optimum, these results still support the high estimate quality of PBTS. One case where PBTS shows a worse performance is the LIGO worfkflow which contains lots of tasks whose execution time is longer than the length of the time-partition (X_P). PBTS can fail to find the optimal start time of such long tasks since PBTS considers only a single time-partition once the spare time distribution is done. By contrast, BTS finds the best schedule across the entire time range between 1 and X_D.

We also measure the time overhead of large workflows with up to 100,000 tasks. Even though the results are not presented in this paper, PBTS spends up to 10 s per time-partition for estimating the resource capacity. Since applications experience only one iteration delay when a workflow starts, the time overhead is negligible in total running time.

5.3. Synthetic workflow

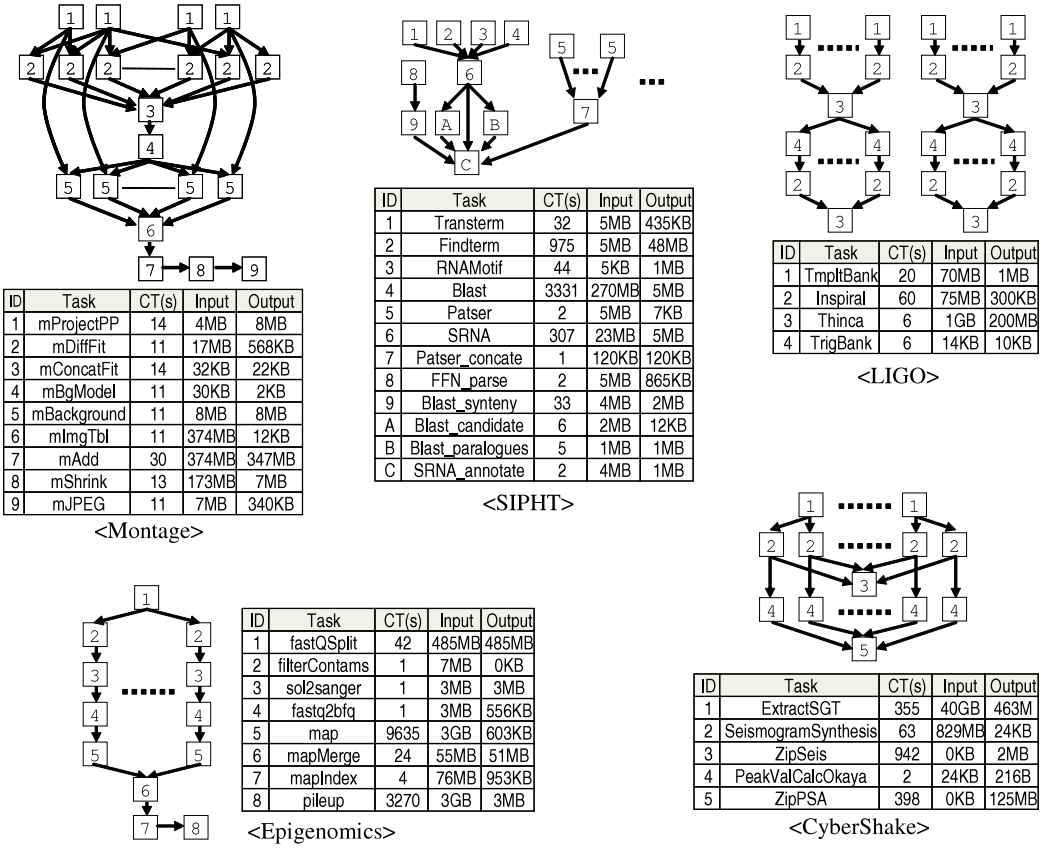
Since the five applications discussed in the previous section cannot cover the characteristics of all scientific applications, we evaluate our algorithm more rigorously against complex workflows with various structures. We classify these synthetic workflows into two groups: *unstructured workflows* and *leveled parallel workflows*.

We use six parameters to define an unstructured workflow: number of tasks, number of edges, range of execution time, range of host requirements for R-task, and percentage of M-tasks. Each edge connects two randomly selected distinct tasks, and the execution time and host requirement of each task are selected via random trials with a uniform distribution over the range. The ranges of parameters are selected to cover as many cases as possible.

By contrast, a leveled parallel workflow has a fixed structure where tasks are divided into several levels and only tasks in adjacent levels can have dependencies. We use four parameters to synthesize a workflow: number of tasks, number of levels, range of the number of tasks in a level, and range of execution time of task. All tasks in the i-th level are parents of tasks in the $(i+1)$-th level. The execution time of a task is selected via random trials as in unstructured workflows. This represents the typical structure of distributed applications such as MapReduce and Dryad.

Through experiments with numerous synthetic workflows, we have observed common trends, and this section presents the representative ones.

We categorize unstructured workflows into three groups according to the existence of M-tasks and the range of HR. First, Fig. 15 shows the average cost of 100 unstructured synthetic workflows without M-task where every task needs a single computing host. The results show that PBTS is consistently better than BTS and BTSE by about 1%–4% while both PBTS and BTSE are close to the optimum. In addition, the cost decreases with the number of time-partitions (X_D/X_P). Second, Fig. 16 shows the average cost of 100 unstructured synthetic workflows without



<Montage>

ID	Task	CT(s)	Input	Output
1	mProjectPP	14	4MB	8MB
2	mDiffFit	11	17MB	568KB
3	mConcatFit	14	32KB	22KB
4	mBgModel	11	30KB	2KB
5	mBackground	11	8MB	8MB
6	mImgTbl	11	374MB	12KB
7	mAdd	30	374MB	347MB
8	mShrink	13	173MB	7MB
9	mJPEG	11	7MB	340KB

<SIPHT>

ID	Task	CT(s)	Input	Output
1	Transterm	32	5MB	435KB
2	Findterm	975	5MB	48MB
3	RNAMotif	44	5KB	1MB
4	Blast	3331	270MB	5MB
5	Patser	2	5MB	7KB
6	SRNA	307	23MB	5MB
7	Patser_concate	1	120KB	120KB
8	FFN_parse	2	5MB	865KB
9	Blast_synteny	33	4MB	2MB
A	Blast_candidate	6	2MB	12KB
B	Blast_paralogues	5	1MB	1MB
C	SRNA_annotate	2	4MB	1MB

<LIGO>

ID	Task	CT(s)	Input	Output
1	TmpltBank	20	70MB	1MB
2	Inspiral	60	75MB	300KB
3	Thinca	6	1GB	200MB
4	TrigBank	6	14KB	10KB

<Epigenomics>

ID	Task	CT(s)	Input	Output
1	fastQSplit	42	485MB	485MB
2	filterContams	1	7MB	0KB
3	sol2sanger	1	3MB	3MB
4	fastq2bfq	1	3MB	556KB
5	map	9635	3GB	603KB
6	mapMerge	24	55MB	51MB
7	mapIndex	4	76MB	953KB
8	pileup	3270	3GB	3MB

<CyberShake>

ID	Task	CT(s)	Input	Output
1	ExtractSGT	355	40GB	463M
2	SeismogramSynthesis	63	829MB	24KB
3	ZipSeis	942	0KB	2MB
4	PeakValCalcOkaya	2	24KB	216B
5	ZipPSA	398	0KB	125MB

Fig. 13. Workflow structures of five applications with the information on computation time (CT) and input/output data size of each task.

M-tasks in which the HR of each task is 2^n where n is selected by a random trial with a uniform distribution between 1 and 7. Likewise, PBTS is consistently better than BTS and BTSE, and the cost decreases as the length of the time-partition decreases. Unlike the static algorithm, both PBTS and BTSE leverage the elasticity of resources accordingly. Furthermore, PBTS saves the cost more aggressively than BTSE and saves about 10% more than BTS. Typically, workflows with multi-HR tasks have a wide fluctuation of resource requirement, and PBTS adapts to the fluctuation better than BTS. Finally, Fig. 17 shows the average cost of 100 unstructured synthetic workflows where the half of tasks are M-tasks. The result is similar to that of the second group except that the cost gap between BTSE and PBTS becomes narrow and that both estimates for large deadlines approach the optimal. The improvement of BTSE is mainly because the flexibility of M-task contributes to reducing the peak capacity.

Since the minimum makespan of an application may not be aligned with charge time unit, users may tolerate some delays of execution in order to fully use the allocated resources. We extend the deadline and observe the effects on the resource cost. The overall trend is that cost decreases gradually to a certain point as deadline increases. However, the cost saving gets saturated beyond an application-specific threshold because the increase of resource allocation time cancels out the gain due to the reduction of resource count so we cannot expect any further savings any more. Actually, the excessive increase of the deadline is not desirable because long running applications are more likely to experience failures.

Then, the question would be how long can we extend the deadline. An interesting observation is that we can save significant cost by slightly extending the deadline beyond the minimum makespan. The results when the deadline is extended to 120% of the minimum makespan are shown in Figs. 15–17. As a rule of thumb, the cost saving is noticeable until the deadline increases up to 120% of the workflow's minimum makespan in most types of synthetic workflows. In practice, we can align the deadline to the boundary of the charge time unit around the 120% extended time.

Lastly, we evaluate PBTS and other approaches with synthetic leveled parallel workflows. Fig. 18 shows the average cost of 100 synthetic workflows with 10 levels. In each level, the number of tasks are randomly selected by a uniform distribution between 10 and 100. The three graphs lead us to the conclusion that PBTS is the most efficient in terms of cost. In addition, the longer deadlines get, the wider the difference between PBTS and BTSE becomes.

5.4. Effect of the runtime feature of PBTS

The experiments in the previous sections assume that the execution time of tasks is accurate and constant. However, the actual execution time can fluctuate at runtime. As a result, resources can be wasted in case of over-provisioning, or the application can miss the deadline otherwise. We simulate the real execution time by using a simplified model and observe how well PBTS can adapt to such dynamics.

The first experiments show how well the PBTS algorithm with the runtime information can save the cost when the actual execution times of tasks are shorter than the predicted ones. This configuration represents the case when allocated resources are mostly faster than user's requirements. We compare this version of the PBTS algorithm to the pure PBTS algorithm without any runtime information. Table 1 summarizes the results for several workflows when the deadline is 120% of the minimum makespan. The task execution time follows a normal distribution where the mean is the normalized actual execution time based on the predicted time; we use two mean values, 0.8 and 0.9, and two

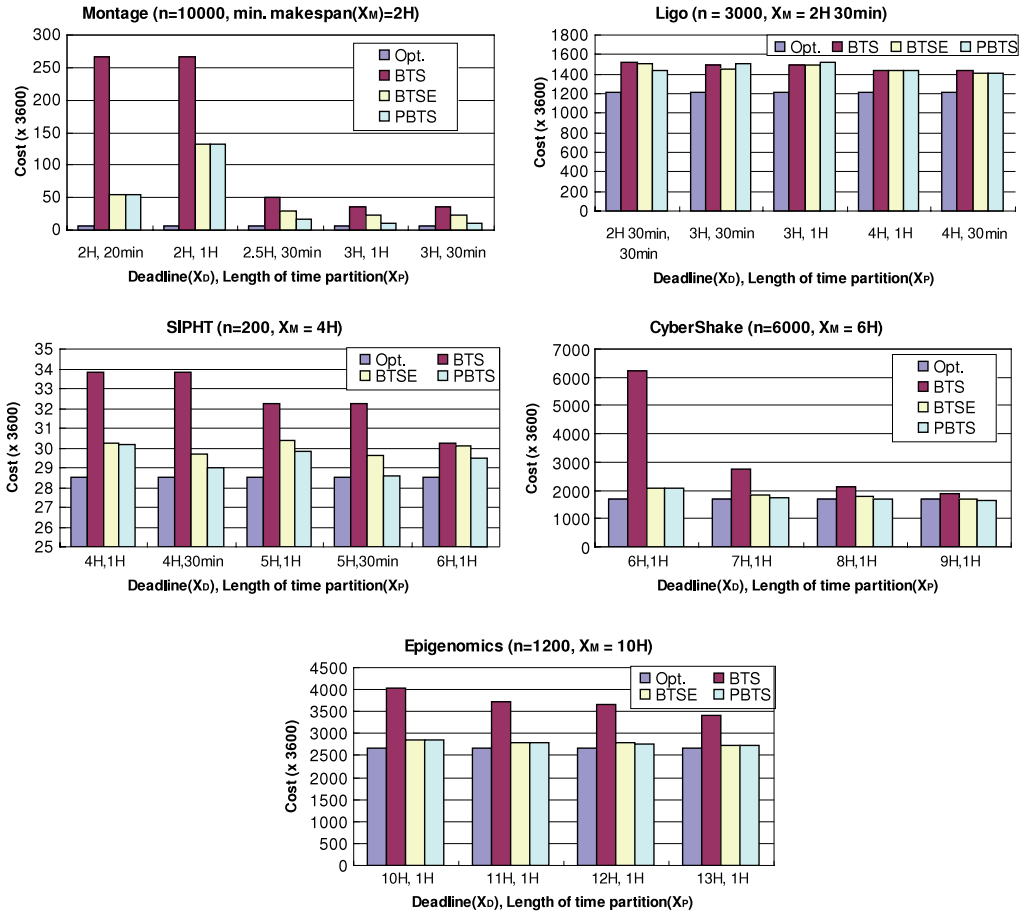


Fig. 14. Comparison of the costs of optimum, BTS, BTSE and PBTS for the five real application workflows.

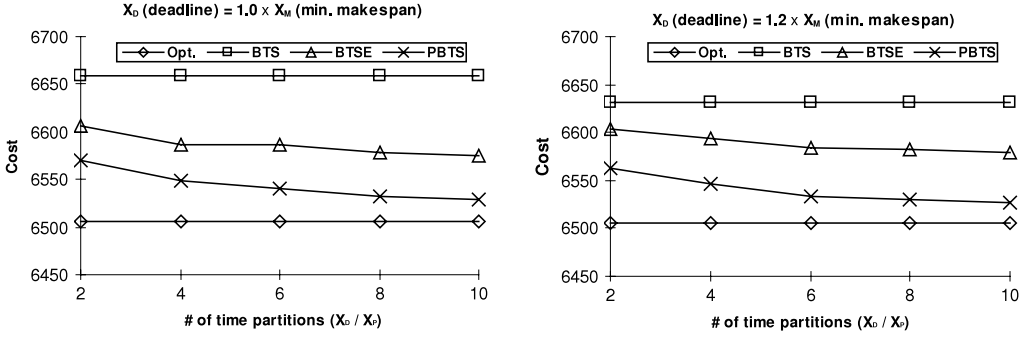


Fig. 15. Comparison of the costs of optimum, BTS, and PBTS for unstructured synthetic workflows without any *M*-tasks and multi-HR *R*-tasks. Average of 100 random workflows with 1000 tasks and 4000 edges.

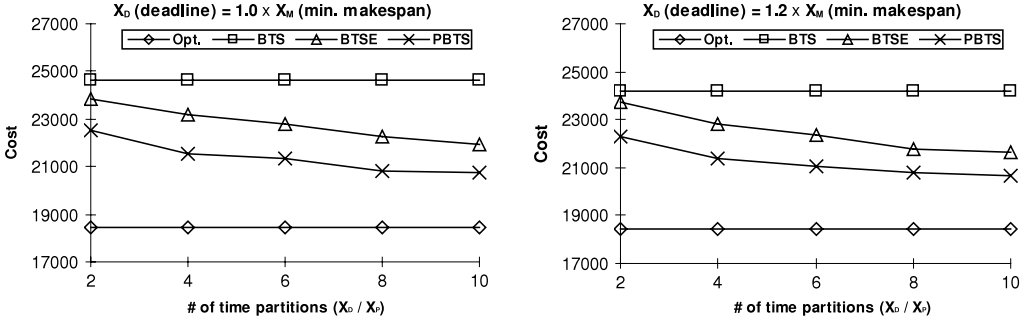


Fig. 16. Comparison of the costs of optimum, BTS, and PBTS for unstructured synthetic workflows without any *M*-tasks. The range of HR is 1–128. Average of 100 random workflows with 100 tasks and 500 edges.

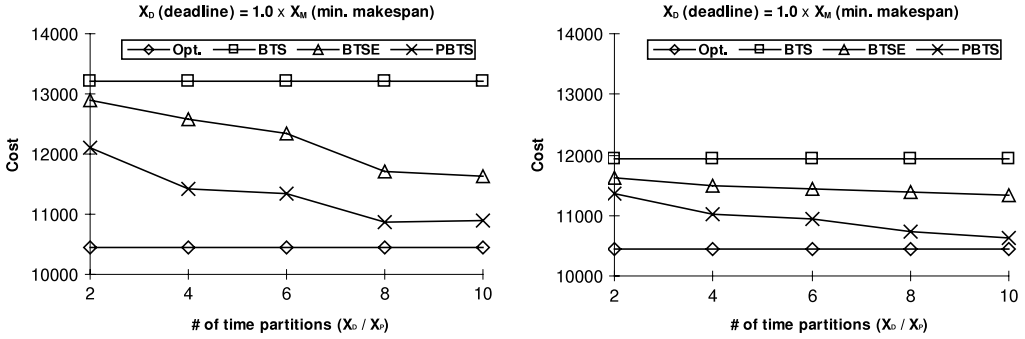


Fig. 17. Comparison of the costs of optimum, BTS, and PBTS for unstructured synthetic workflows with 50% of M-tasks. Average of 100 random workflows with 200 tasks and 500 edges.

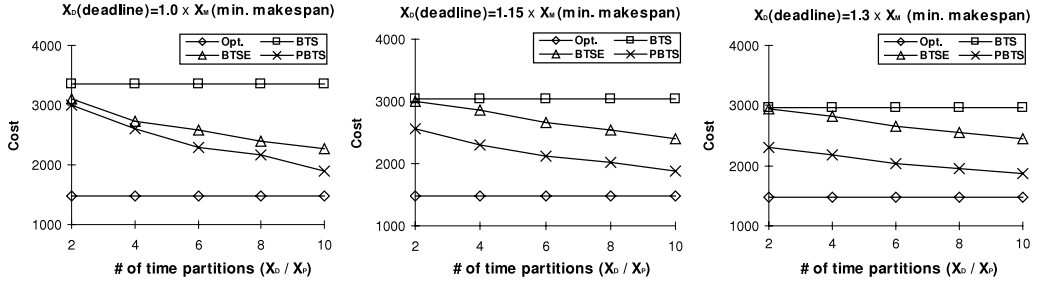


Fig. 18. Comparison of the costs of optimum, BTS, and PBTS for leveled parallel synthetic workflows. Average of 100 random workflows with 10 levels and 10–100 tasks in each level.

Table 1
Relative costs of PBTS with runtime information compared to PBTS without runtime feature (mean values (μ) and standard deviations (σ) are the ratio between actual execution time and predicted execution time).

	$X_D/X_P = 4$		$X_D/X_P = 8$	
Workflows	$\mu = 0.9, \sigma = 0.1$ (%)	$\mu = 0.8, \sigma = 0.2$ (%)	$\mu = 0.9, \sigma = 0.1$ (%)	$\mu = 0.8, \sigma = 0.2$ (%)
Montage	88.5	83.3	84.2	81.8
LIGO	94.2	91.3	92.5	87.3
SIPHT	95.2	91.9	91.6	89.8
CyberShake	90.2	85.7	88.3	82.4
Epigenomics	89.2	86.7	85.9	83.1

standard deviation values, 0.1 and 0.2. The values in the table are the ratio of the cost of PBTS with runtime information to that of pure PBTS. PBTS successfully reduces the overall cost of the application, exploiting the slack time of faster tasks. This also means that PBTS can take advantage of faster resources. Moreover, the results show that PBTS can get more cost savings with finer time-partitions since PBTS can use more up-to-date information and have more chances to adjust its estimate.

We repeat the same experiments when the actual execution times of tasks are longer than the predicted ones. Table 2 shows the results with a mean of 1.2 and a standard deviation of 0.2 of the predicted ET. PBTS fails to meet the deadline even with the runtime information because PBTS cannot handle the cases when the sum of execution times of tasks on the critical path is longer than the deadline.

The online characteristic of PBTS allows us to add features to reduce the deadline miss probability whereas it is not easy with static approaches such as BTS and traditional scheduling algorithms. For example, PBTS can selectively adopt a redundant execution scheme for tasks on critical path. Besides, we can apply PBTS more conservatively with longer execution time of tasks (e.g. worst case execution time). Then, this approach becomes equivalent to the case discussed in Table 1 since the actual execution times are likely shorter than the predicted ones. PBTS can reduce the deadline miss probability at the small expense of the cost.

Table 2
Actual turnaround time of workflow compared to the deadline (120% of the minimum makespan) when the actual execution time of each task follows a normal distribution (μ = 120% of the predicted execution time, standard deviation = 20% of the predicted execution time).

Workflows	w/o runtime info. (%)	With runtime info. (%)
Montage	116.08 ± 6.58	112.52 ± 7.72
LIGO	125.22 ± 9.21	123.74 ± 10.51
SIPHT	114.52 ± 6.06	111.34 ± 8.84
CyberShake	116.82 ± 9.57	112.51 ± 10.89
Epigenomics	123.14 ± 7.33	120.85 ± 9.47

Especially the runtime characteristic enables PBTS to save more cost than BTSE. We measure the cost when both PBTS and BTES set the predicted execution time of every task by 20% longer than the values given by users while the actual execution time follows a normal distribution with a mean value of 100% and a standard deviation of 20% of the given values and the deadline is 120% of the minimum makespan (X_M). We have observed that this approach can reduce the deadline miss probability down to 20% or less. The relative cost of BTSE to BTS shown in Table 3 proves that PBTS can save about 20% cost compared to BTSE when the predicted execution time is mostly longer than the actual execution time. We plan to leverage such inline characteristic of PBTS in order to make the algorithm more robust and tolerant of runtime dynamics in our future work.

Table 3
The relative costs of BTSE to PBTS when both set the predicted execution time of every task by 20% longer than the values given by users while actual execution time follows a normal distribution ($\mu = 100\%$ and $\sigma = 20\%$ of the given values, $X_D = 1.2X_M$).

Workflows	$Cost_{BTSE}/Cost_{PBTS}$
Montage	1.218
LIGO	1.125
SIPHT	1.190
CyberShake	1.235
Epigenomics	1.107
Unstructured synthetic	1.380
Leveled parallel synthetic	1.182

6. Conclusion

In this paper, we have suggested an architecture for executing workflow applications on elastic computing resources. In addition, we have proposed the core algorithm named PBTS for estimating the minimum resource capacity to execute a workflow within a given deadline. The main contribution of our research is that PBTS bridges the gap between workflow management system and the state-of-the-art resource provisioning environments. With this framework, scientists are able to exploit cloud while saving more operational cost with utility basis billing policy. PBTS estimates resource capacity per time-partition so that it minimizes the resource cost, satisfying the deadlines. In addition, PBTS handles any workflows of DAG structure that includes single, data-parallel, and even MPI-like tasks. Finally, PBTS determines not only the resource capacity but also the time schedule and the host requirements of tasks.

Various experiments with synthetic and real workflows have demonstrated that the PBTS algorithm performs better than the alternative approaches in terms of cost, and its performance is close to the theoretical low bound. In addition, the algorithm has a polynomial time complexity, taking only a few seconds even for large workflows. Moreover, we have shown that users can benefit from the relaxation of deadline to trade the execution time for cost savings; as a rule of thumb, the deadline can be extended up to the 120% of the minimum makespan of workflow. Finally, PBTS can leverage the runtime information of tasks. PBTS can adapt to the fluctuation of execution time of individual tasks and meet the deadline of the application successfully in most cases.

We plan to improve PBTS in several directions. First, even though PBTS can adapt to its estimate according to the dynamics of tasks, it may fail if the tasks on the critical path are delayed too much or in the worst cases if some tasks crash. The bottom line is that PBTS can use the runtime information and adjust the resource capacity accordingly on the fly. PBTS can use various options; tasks can be executed redundantly based on the task failure probability, or the worst case execution time can be used as the predicted time. Second, PBTS does not consider I/O contention when determining the start time of task. Many workflow applications are data-intensive, and load/store bandwidth can be an issue. Finally, we assume that a workflow has a DAG structure. However, a workflow can have loops or conditional branches as in BPEL [43] or Spark [44], or tasks can be added dynamically at runtime.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MEST) (No. 2010-0026511).

References

[1] J.C. Jacob, D.S. Katz, T. Prince, The montage architecture for Grid-enabled science processing of large, distributed datasets, in: Proceedings of the 4th Earth Science Technology Conference, ESTC2004, 2004.
[2] S.J. Ludtke, P.R. Baldwin, W. Chiu, EMAN: semiautomated software for high-resolution single-particle reconstructions, Journal of Structural Biology 128 (1999) 82–97.
[3] B. Plale, et al., CASA and LEAD: adaptive cyberinfrastructure for real-time multiscale weather forecasting, IEEE Computer 39 (2006) 56–64.
[4] H. Magistrale, S. Day, R.W. Clayton, R. Graves, The SCEC southern California reference three-dimensional seismic velocity model version 2, Bulletin of the Seismological Society of America 90 (2000) S65–S76.
[5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, 2004, pp. 137–150.
[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, SIGOPS Operating Systems Review 41 (3) (2007) 59–72.
[7] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, D. Katz, Pegasus: a framework for mapping complex scientific workflows onto distributed systems, Scientific Programming Journal 13 (3) (2005) 219–237.
[8] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, J.C. Seragiotto, H.-L. Truong, Askalon: a tool set for cluster and Grid computing, Concurrency and Computation: Practice and Experience 17 (2–4) (2005).
[9] I. Taylor, M. Shields, I. Wang, A. Harrison, Visual Grid workflow in Triana, Journal of Grid Computing 3 (3–4) (2005) 153–169.
[10] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, K. Yocumet, Sharing networked resources with brokered leases, in: Proceedings of the USENIX Technical Conference, 2006.
[11] Y.-S. Kee, K. Yocum, A.A. Chien, H. Casanova, Improving Grid resource allocation via integrated selection and binding, in: Proceedings of the 19th ACM/IEEE International Conference on High Performance Computing and Communication, 2006.
[12] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, A.A. Chien, Efficient resource description and high quality selection for virtual Grids, in: Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid, 2005.
[13] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The Eucalyptus open-source cloud-computing system, in: Proceedings of the 1st Workshop on Cloud Computing and its Applications, 2008.
[14] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2.
[15] E.-K. Byun, Y.-S. Kee, E. Deelman, K. Vahi, G. Mehta, J.-S. Kim, Estimating resource needs for time-constrained workflows, in: Proceedings of the 4th IEEE International Conference on e-Science, 2008.
[16] E.-K. Byun, Y.-S. Kee, J.-S. Kim, E. Deelman, S. Maeng, Bts: resource capacity estimate for time-targeted science workflows, Journal of Parallel and Distributed Computing 71 (6) (2011) 848–862.
[17] Y.-S. Kee, E.-K. Byun, E. Deelman, K. Vahi, J.-S. Kim, Pegasus on the virtual Grid: a case study of workflow planning over captive resources, in: Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science, 2008.
[18] NCSA TeraGrid linux cluster. http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/TGIA64LinuxCluster.
[19] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.H. Su, K. Vahi, Characterization of scientific workflows, in: Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science, 2008.
[20] A. Radulescu, A. van Gemund, A low-cost approach towards mixed task and data parallel scheduling, in: Proceedings of the 2001 International Conference on Parallel Processing, 2001.
[21] R. Shankar, S. Sachin, B. Prithviraj, A framework for exploiting task and data parallelism on distributed memory multicomputers, IEEE Transactions on Parallel and Distributed Systems 8 (11) (1997) 1098–1116.
[22] R. Thomas, R. Gudula, Compiler support for task scheduling in hierarchical execution models, Journal of Systems Architecture 45 (6–7) (1999) 483–503.
[23] Y.K. Kwok, I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, Journal of Parallel and Distributed Computing 59 (3) (1999) 381–422.
[24] J. Yu, R. Buyya, K. Ramanohanarao, Metaheuristics for Scheduling in Distributed Computing Environments, Springer, Berlin, Germany, 2008.
[25] F. Dong, S.G. Akl, Scheduling algorithms for Grid computing: state of the art and open problems, Tech. Rep., School of Computing, Queen's University, Kingston, Ontario, January 2006.
[26] H. Topcuoglu, S. Hariri, M. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274.
[27] T.L. Adam, K.M. Chandy, J.R. Dickson, A comparison of list schedules for parallel processing systems, Communications of the ACM 17 (12) (1974) 685–690.
[28] G.C. Shih, E.A. Lee, A Compile-Tim scheduling heuristics for interconnection-constrained heterogeneous processor architecture, IEEE Transactions on Parallel and Distributed Systems 4 (2) (1993) 75–87.
[29] M. Rahman, S. Venugopal, R. Buyya, A dynamic critical path algorithm for scheduling scientific workflow applications on global Grids, in: Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing, 2007.
[30] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, P. Maechling, Scientific workflow applications on Amazon EC2, in: Proceedings of the Workshop on Cloud-based Services and Applications in conjunction with the 5th IEEE International Conference on e-Science, e-Science 2009, 2009.
[31] C. Evangelinos, C. Hill, Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2, in: Proceedings of the 1st Workshop on Cloud Computing and its Applications, 2008.

[32] Science clouds. http://www.scienceclouds.org/.
[33] The nimbus toolkit. http://www.nimbusproject.org/.
[34] Amazon elastic mapreduce. http://aws.amazon.com/elasticmapreduce.
[35] M.A. Murphy, B. Kagey, M. Fenn, S. Goasguen, Dynamic provisioning of virtual organization clusters, in: Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid, 2009.
[36] P. Marshall, K. Keahey, T. Freeman, Elastic site: using clouds to elastically extend site resources, in: Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2010.
[37] A. Sudarsanam, M. Srinivasan, S. Panchanathan, Resource estimation and task scheduling for multithreaded reconfigurable architectures, in: Proceedings of the 10th International Conference on Parallel and Distributed Systems, 2004.
[38] M. Wieczorek, S. Podlipnig, R. Prodan, T. Fahringer, Bi-criteria scheduling of scientific workflows for the Grid, in: Proceedings of the 8th ACM/IEEE International Symposium on Cluster Computing and the Grid, 2008.
[39] R. Huang, H. Casanova, A.A. Chien, Automatic resource specification generation for resource selection, in: Proceedings of the 20th ACM/IEEE International Conference on High Performance Computing and Communication, 2007.
[40] Usc epigenomic center. http://epigenome.usc.edu.
[41] D.A. Brown, P.R. Brady, A. Dietz, J. Cao, B. Johnson, J. McNabb, A case study on the use of workflow technologies for scientific analysis: gravitational wave data analysis, in: Workflows for e-Science, Springer, 2006 (Chapter).
[42] J. Livny, H. Teonadi, M. Livny, M.K. Waldor, High-throughput, kingdom-wide prediction and annotation of bacterial non-coding RNAs, PLoS ONE 3 (9) (2008).
[43] Web services business process execution language, version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
[44] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: The 2nd USENIX Workshop on Hot Topics in Cloud Computing, 2010.

Eun-Kyu Byun received his B.S. and M.S. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2003 and 2005, respectively. Currently, he is pursuing his Ph.D. degree in Computer Science at the same school. His research interests include distributed systems, cloud computing, resource management, and workflow management.

Yang-Suk Kee (Yang Seok Ki) is a Senior Member of Technical Staff at Oracle America and involved in designing and developing the next generation of the Oracle Streams Advanced Queuing (AQ) system. His research interest spans high performance scientific/enterprise computing (HPC), message oriented middleware (MOM), service oriented architecture (SOA), Grid/cloud computing, parallel computing. Before Dr. Kee joined Oracle, he was a post-doctoral researcher under Dr. Carl Kesselman at Information Sciences Institute, University of Southern California and under Dr. Andrew A. Chien and Dr. Henri Cassanova at University of California, San Diego. Dr. Kee actively participated in the VGrADS (Virtual Grid Application Development System) project as a leading core contributor to the VGES (Virtual Grid Execution System). He was a lecturer of Graduate Study at Seoul National University and lead the ParADE (Parallel Application Development Environment) and xBSP (eXpress Bulk Synchronous Parallel) projects. He received Ph.D. in Electrical Engineering and Computer Science, Master of Computer Engineering, and Bachelor degrees of Computer Engineering from Seoul National University.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor in Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Seungryoul Maeng received the B.S. degree in Electronics Engineering from Seoul National University, Korea, in 1977, and the M.S. and Ph.D. degrees in Computer Science from KAIST in 1979 and 1984, respectively. Since 1984 he has been a faculty member at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include computer architecture, cluster computing, and embedded systems.