Budget-Driven Scheduling of Scientific Workflows in IaaS Clouds with Fine-Grained Billing Periods

MARIA A. RODRIGUEZ and RAJKUMAR BUYYA, The University of Melbourne

With the advent of cloud computing and the availability of data collected from increasingly powerful scientific instruments, workflows have become a prevailing mean to achieve significant scientific advances at an increased pace. Scheduling algorithms are crucial in enabling the efficient automation of these large-scale workflows, and considerable effort has been made to develop novel heuristics tailored for the cloud resource model. The majority of these algorithms focus on coarse-grained billing periods that are much larger than the average execution time of individual tasks. Instead, our work focuses on emerging finer-grained pricing schemes (e.g., per-minute billing) that provide users with more flexibility and the ability to reduce the inherent wastage that results from coarser-grained ones. We propose a scheduling algorithm whose objective is to optimize a workflow's execution time under a budget constraint; quality of service requirement that has been overlooked in favor of optimizing cost under a deadline constraint. Our proposal addresses fundamental challenges of clouds such as resource elasticity, abundance, and heterogeneity, as well as resource performance variation and virtual machine provisioning delays. The simulation results demonstrate our algorithm's responsiveness to environmental uncertainties and its ability to generate high-quality schedules that comply with the budget constraint while achieving faster execution times when compared to state-of-the-art algorithms.

CCS Concepts: • **Computing methodologies → Distributed algorithms;**

Additional Key Words and Phrases: Budget, IaaS cloud, makespan minimisation, scientific workflow, scheduling, resource provisioning

ACM Reference Format:
Maria A. Rodriguez and Rajkumar Buyya. 2017. Budget-driven scheduling of scientific workflows in IaaS clouds with fine-grained billing periods. ACM Trans. Auton. Adapt. Syst. 12, 2, Article 5 (May 2017), 22 pages.
DOI: http://dx.doi.org/10.1145/3041036

1. INTRODUCTION

Scientific workflows describe a series of computations that enable the analysis of data in a structured and distributed manner. They have been successfully used to make significant scientific advances in various fields such as biology, physics, medicine, and astronomy [Gil et al. 2007]. Their importance is exacerbated in today's big data era, as they become a compelling mean to process and extract knowledge from the ever-growing data produced by increasingly powerful tools such as telescopes, particle accelerators, and gravitational wave detectors. Due to their large-scale, data, and compute-intensive nature, scheduling algorithms are key to efficiently automating their execution in distributed environments and, as a result, to facilitating and accelerating the pace of scientific progress.

Authors' addresses: M. A. Rodriguez and R. Buyya, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Parkville VIC 3010, Australia; emails: {marodriguez, rbuyya}@unimelb.edu.au.

DOI: http://dx.doi.org/10.1145/3041036

The emergence of cloud computing has brought with it several advantages for the deployment of large-scale scientific workflows. In particular, Infrastructure as a Service (IaaS) clouds allow workflow management systems to access a virtually infinite pool of resources that can be acquired, configured, and used as needed and are charged on a pay-per-use basis. IaaS providers offer virtualized compute resources called Virtual Machines (VMs) for lease. They have a predefined CPU, memory, storage, and bandwidth capacity, and different resource bundles (i.e., VM types) are available at varying prices. They can be elastically acquired and released and are generally charged per time frame or billing period. While VMs deliver the compute power, IaaS clouds also offer storage and networking services, providing the necessary infrastructure for the execution of workflow applications.

The adoption of cloud computing for scientific workflow deployment has led to extensive research on designing efficient scheduling algorithms capable of elastically utilizing VMs. This ability to modify the underlying execution environment is a powerful tool that allows algorithms to scale the number of resources to achieve both performance and cost efficiency. However, this flexibility is limited when coarse-grained billing periods such as hourly billing are enforced by providers. As the average execution time of workflow tasks is considerably smaller than a billing cycle, algorithms are obliged to focus on maximizing the usage of time slots in leased VMs as a cost-controlling mechanism. This not only restricts the degree of scalability in terms of resources but also leads to inevitable wastage as idle time slots will naturally occur due to performance restrictions and dependencies between tasks. This coarse-grained billing period is assumed by the majority of existing algorithms dealing with resource provisioning and scheduling in clouds. Instead, our work targets emerging pricing models that are designed to give users more flexibility and reduce wastage by offering fine-grained charging periods such as per-minute billing. Under this model, algorithms can more freely take advantage of the cloud's resource abundance, and, as a result, more aggressive dynamic scaling policies are needed. The potential of using a different VM for each workflow task emphasizes the importance of making accurate resource provisioning decisions that are not only guided by the scheduling objectives but also by characteristics inherent to clouds such as resource performance variation and a non-negligible VM provisioning delay.

The utility-based pricing model offered by cloud providers means that finding a trade-off between cost and performance is a common denominator for scheduling algorithms. This is done mostly by trying to minimize the total infrastructure cost while meeting a time constraint or deadline. Only a small fraction of techniques focus on scheduling under budget constraints. Most of them are based on computationally intensive metaheuristic techniques that do not scale well with the number of tasks in the workflow and that produce a static schedule unable to adapt to the inherent dynamicity of cloud environments. Others include a deadline constraint that guides the optimization process, and the budget is only taken into consideration when deciding the feasibility of a potential schedule. Contrary to this, we propose a budget-driven algorithm whose objective is to optimize the way in which the budget is spent so that the makespan (i.e., total execution time) of the application is minimized. It includes a budget distribution strategy that guides the individual expenditure on tasks and makes dynamic resource provisioning and scheduling decisions to adapt to changes in the environment. Also, to improve the quality of the optimization decisions made, two different mathematical models are proposed to estimate the optimal resource capacity for parallel tasks derived from data distribution structures. Our simulation results demonstrate that our algorithm is scalable, adaptable, and capable of generating efficient schedules with high quality in terms of meeting the budget constraint with lower makespans when compared to state-of-the-art algorithms.

The rest of this article is organized as follows. Section 2 presents the related work followed by the application and resource models in Section 3. Section 4 explains the proposed resource provisioning and scheduling algorithm. Section 5 presents the experimental setup and the evaluation of our solution. Finally, conclusions and future work are outlined in Section 6.

2. RELATED WORK

Our work is related to algorithms for workflow scheduling in IaaS clouds capable of elastically scaling resources. The Partitioned Balanced Time Scheduling (PBTS) algorithm [Byun et al. 2011] divides the execution of the workflow into time partitions the size of the billing period. Then, it optimizes the schedule of the tasks in each partition by estimating the minimum number of homogeneous VMs required to finish them on time. It differs from our solution in that we do not assume tasks can finish within one billing period, and we consider VM types with different characteristics and prices. SCS [Mao and Humphrey 2011] and Dyna [Zhou et al. 2016] are other algorithms with an auto-scaling mechanism to dynamically allocate and deallocate VMs based on the current status of tasks. They differ from our proposal as they consider dynamic and unpredictable workloads of workflows and assume an hourly billing period. Designed to schedule a single workflow while dynamically making resource provisioning decisions are the heuristics proposed by Poola et al. [2014] and Wang et al. [2014]; however, they also assume a pricing model based on an hourly rate. Furthermore, all of the mentioned algorithms have different objectives to our solution, as they aim to minimize the execution cost while meeting a deadline constraint.

The Dynamic Provisioning Dynamic Scheduling (DPDS) algorithm [Malawski et al. 2012] is another strategy that dynamically scales the VM pool and was designed to schedule a group of interrelated workflows (i.e., ensembles) under budget and deadline constraints. It does so by creating an initial pool of homogeneous VMs with as many resources as allowed by the budget and updating it at runtime based on a utilization measure estimated using the number of busy and idle VMs. DPDS differs from our work in several aspects; mainly, its provisioning strategy is suited for coarse-grained periods and we focus on scheduling a single workflow without considering a deadline constraint.

Only a few algorithms targeting IaaS clouds consider a budget constraint as part of their objectives. The Static Provisioning Static Scheduling (SPSS) algorithm [Malawski et al. 2012] considers the scheduling of workflow ensembles under deadline and budget constraints. The deadline guides the scheduling process of individual workflows by assigning sub-deadlines to tasks. These are then assigned to VMs that can complete their execution on time with minimum cost. This process is repeated until all the workflows have been scheduled or the budget has been reached. Pietri et al. [2013] proposed SPSS-EB, an algorithm based on SPSS and concerned with meeting energy and budget constraints. The execution of the workflow is planned by scheduling each task so the total energy consumed is minimum, a plan is then accepted an executed only if the energy and budget constraints are met. Our work is different from these approaches in two aspects. First, they consider a second constraint as part of the scheduling objectives. Moreover, they make static provisioning and scheduling decisions and do not account for VM provisioning and deprovisioning delays or performance degradation.

A dynamic budget-aware algorithm capable of making auto-scaling and scheduling decisions to minimize the application's makespan is presented by Mao and Humphrey [2013]. However, they consider an hourly budget instead of a budget constraint for the entire workflow execution and aim to optimize the execution of a continuous workload of workflows. Similarly to our work, the Critical-Greedy [Wu et al. 2015] algorithm considers a financial constraint while minimizing the end-to-end delay of the workflow execution. However, it does not include billing periods on its cost estimates and hence

considers VMs priced per unit of time. Also, the output of the algorithm is a task to VM type mapping and the authors do not propose a strategy to assign the tasks to actual VMs while considering their startup time and performance degradation.

The Revised Discrete Particle Swarm Optimization (RDPSO) algorithm [Wu et al. 2010] uses a technique based on particle swarm optimization to produce a near-optimal schedule that minimizes either cost or time and meets constraints such as deadline and budget. In contrast to our approach, the algorithm is based on a computationally intensive meta-heuristic technique that produces a globally optimized schedule. ScaleStar [Zeng et al. 2012] is another algorithm that considers a budget constraint. Similarly to our approach, it aims to minimize the makespan of the workflow. However, although it explicitly considers billing periods of one hour, their total execution cost calculation does not consider the fact that partial utilization of VMs is charged as full time utilization. These algorithms also differ from our solution in that they produce static schedules and assume a finite set of VMs is available as input.

Malawski et al. [2015] present a mathematical model that optimizes the cost of scheduling workflows under a deadline constraint. As opposed to our algorithm, it considers a multi-cloud environment where each provider offers a limited number of VMs billed per hour. They group tasks on each level based on their computational cost and input/output data and schedule these groups instead of single tasks. They achieve this by modeling the problem as a mixed integer program, which differs from ours as it generates a static schedule for the entire workflow as opposed to a resource provisioning plan for a subset of the workflow tasks. Genez et al. [2012] also formulate the problem of scheduling a workflow on a set of subscription-based and on-demand instances as an integer program. However, the output of their model is a static schedule indicating the mapping of tasks to VMs as well as the time when they are meant to start their execution. This limits the scalability of the algorithm, as the number of variables and constraints in the formulation increases rapidly with the number of cloud providers, maximum number of VMs that can be leased from each provider, and the number of tasks in the workflow.

3. APPLICATION AND RESOURCE MODELS

This work is designed to schedule scientific workflows composed of tasks that are computationally and data intensive. Specifically, we consider workflows modeled as Directed Acyclic Graphs (DAGs); that is, graphs with directed edges and no cycles or conditional dependencies. Formally, a workflow W is composed of a set of tasks $T = \{t_1, t_2, \ldots, t_n\}$ and a set of edges E. An edge $e_{ij} = (t_i, t_j)$ exists if there is a data dependency between t_i and t_j, case in which t_i is said to be the parent task of t_j and t_j the child task of t_i. Based on this, a child task cannot run until all of its parent tasks have completed their execution and its input data are available in the corresponding compute resource. The amount of input data required by task t is defined as d_t^{in}, and the amount of output data it produces as d_t^{out}.

We define the sharing of data between tasks to take place via a global storage system such as Amazon S3 [Google 2015a]. In this way, tasks store their output in the global storage and retrieve their inputs from the same. This model has two main advantages. First, the data are persisted and, hence, can be used for recovery in case of failures. Moreover, unlike a peer-to-peer model where VMs need to remain active until all of the child tasks have received the corresponding data, a shared storage enables asynchronous computation as the VM running the parent task can be released as soon as the data are persisted in the storage system.

We assume a pay-as-you go model where VMs are leased on-demand and are charged per billing period τ. We acknowledge that any partial utilization results in the VM usage being rounded up to the nearest billing period. Nonetheless, we focus on

fine-grained billing periods such as one minute, as offered by providers such as Google Compute Engine Google [2015c] and Microsoft Azure Microsoft [2015]. We consider a single cloud provider and a single data center or availability zone. In this way, network delays are reduced and intermediate data transfer fees eliminated. Finally, we impose no limit on the number of VMs that can be leased from the provider.

We acknowledge that characteristics such as multi-tenancy, virtualization, and the heterogeneity of non-virtualized hardware in clouds result in variability in the performance of resources [Schad et al. 2010; Ostermann et al. 2010; Gupta and Milojicic 2011; Iosup et al. 2011; Jackson et al. 2010]. In particular, we assume a variation in the performance of network resources and VM CPUs with their maximum achievable performance being based on the bandwidth and CPU capacity advertised by the provider. Ultimately, this results in a degradation of data transfer times and task execution times.

The IaaS provider offers a range of VM types $VMT = \{vmt_1, vmt_2, \ldots, vmt_n\}$ with different prices and configurations. The execution time, E_t^{vmt}, of each task on every VM type is available to the scheduler. Different performance estimation methods can be used to obtain this value. The simplest approach is to calculate it based on an estimate of the size of the task and the CPU capacity of the VM type. Another valid method could be based on the results obtained after profiling the tasks on a baseline machine. This topic is out of the scope of this article; however, notice that our solution acknowledges that this value is simply an estimate and does not rely on it being 100% accurate to achieve its objectives.

VM types are also defined in terms of their cost per billing period c_{vmt} and bandwidth capacity b_{vmt}. An average measure of their provisioning $prov_{vmt}$ delay is also included as part of their definition. We assume a global storage system with an unlimited storage capacity. The rates at which it is capable of reading and writing are GS_r and GS_w, respectively. The time it takes to transfer and write d output data from a VM of type vmt into the storage is defined as

$$N_{d,vmt}^{out} = (d/b_{vmt}) + (d/GS_w). \tag{1}$$

Similarly, the time it takes to transfer and read a task's output data from the storage to a VM of type vmt is defined as

$$N_{d,vmt}^{in} = (d/b_{vmt}) + (d/GS_r). \tag{2}$$

As depicted in Equation (3), the total processing time PT_t^{vmt} of task t on a VM of type vmt is calculated as the sum of the task's execution time and the time it takes to read the required n_{in} input files from the storage and write n_{out} output files to it. Notice that there is no need to read an input file whenever it is already available in the VM were the task will execute. This occurs when parent and child tasks run on the same machine,

$$P_t^{vmt} = E_t^{vmt} + \left(\sum_{i=1}^{n_{in}} N_{d_i,vmt}^{in}\right) + \left(\sum_{i=1}^{n_{out}} N_{d_i,vmt}^{out}\right). \tag{3}$$

The cost of using a resource r_{vmt} of type vmt for $lease_r$ time units is defined as

$$C_{r_{vmt}} = \lceil (prov_{vmt} + lease_r)/\tau \rceil * c_{vmt}. \tag{4}$$

Finally, we assume data transfers in and out of the global storage system are free of charge, as is the case for products like Amazon S3 [Google 2015a], Google Cloud Storage [Google 2015b], and Rackspace Block Storage [Rackspace 2015]. As for the actual data storage, most cloud providers charge based on the amount of data being stored. We do not include this cost in the total cost calculation of neither our implementation nor the implementation of the algorithms used for comparison in the experiments. The reason for this is to be able to compare our approach with others designed to transfer

files in a peer-to-peer fashion. Furthermore, regardless of the algorithm, the amount of stored data for a given workflow is most likely the same in every case or it is similar enough that it does not result in a difference in cost.

The scheduling problem addressed in this article can then be defined as dynamically scaling a set of resources R and allocating each task to a given resource so the total cost,

$$C_{total} = \sum_{i=1}^{|R|} C_{r_i^{vmt}}, \tag{5}$$

is less than or equal to the workflow's budget β while minimizing the makespan of the application.

4. PROPOSED APPROACH

We propose a budget-driven algorithm called BAGS in which different resource provisioning and scheduling strategies are used for different topological structures. This is done by partitioning the DAG into bags of tasks (BoTs) containing a group of parallel homogeneous tasks, parallel heterogeneous tasks, or a single task. This strategy derives from the observation that large groups of parallel tasks is a common occurrence in scientific workflows, and, as a result, we aim to optimize their execution as an attempt to generate higher-quality schedules while maintaining the dynamicity and adaptability of the algorithm to the underlying cloud environment.

More specifically, our strategy identifies sets of tasks that are at the same level in the DAG and are guaranteed to be ready for execution at the same time. This may happen when they are at the entry level of the workflow and have no parent tasks dictating the time of their execution or when they share a single parent task that distributes data to them. Figure 1 shows examples of these BoTs in five well-known scientific workflows. Any task that does not meet any of the above requirements is categorized as a single task, or, as we will refer to from now on, a bag with a single task. Each BoT is then scheduled using different strategies tailored for its particular characteristics.

Our algorithm consists of four main stages. The first one is an offline strategy that partitions the DAG into BoTs prior to its execution. The second one is an online budget distribution phase repeated throughout the execution of the workflow. It assigns a portion of the remaining budget to the tasks that have not been scheduled yet. The third stage is responsible for creating a resource provisioning plan for BoTs as their tasks become available for execution. Finally, ready tasks are scheduled and executed based on their corresponding provisioning plan. Each of these phases is explained in detail in the following sections.

4.1. DAG Preprocessing

This stage is responsible for identifying and partitioning the DAG into BoTs. Tasks are grouped together if they belong to the same data distribution structure and share the same single parent or if they are entry tasks and have no parent tasks associated with them. If a task does not meet any of these requirements, then it is placed on its own, single-task bag. BoTs with multiple tasks are further categorized into two different classes. The first category is groups of parallel homogeneous tasks, that is, all tasks in the bag are of the same type in terms of the computations they perform. The second one is composed of groups of heterogeneous tasks.

Hence, the preprocessing stage leads to the identification of the following sets:

—$BoT_{hom} = \{bot_1, bot_2, \ldots, bot_n\}$: Set of bags of homogeneous tasks,
—$BoT_{het} = \{bot_1, bot_2, \ldots, bot_m\}$: Set of bags of heterogeneous tasks,
—$BoT_{sin} = \{bot_1, bot_2, \ldots, bot_s\}$: Set of bags containing a single task.

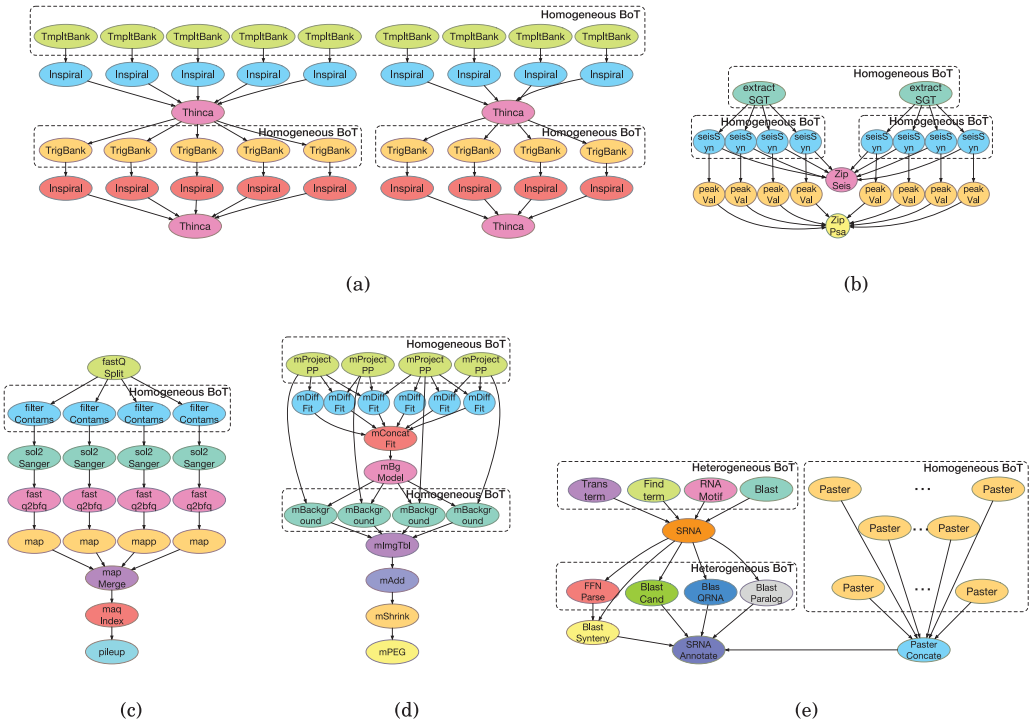


Fig. 1. Examples of BoTs in five well-known scientific workflows. Tasks not belonging to a homogeneous or heterogeneous BoT are classified as a single-task BoT: (a) Ligo, (b) CyberShake, (c) Epigenomics, (d) Montage, and (e) SIPHT.

4.2. Budget Distribution

The budget distribution phase assigns each individual task a portion of the budget and ultimately determines how fast a task can be processed. Although we propose a strategy here, it is worthwhile mentioning that this method can be easily interchanged without altering the methodology of the algorithm. During this stage, the cost of a task on a given VM type is estimated using the following equation:

$$C_t^{vmt} = \lceil P_t^{vmt}/\tau \rceil * c_{vmt}. \tag{6}$$

This definition relies on our assumption of fine-grained billing periods as a task's execution time is likely to be close to a multiple of the billing period, and if there is spare time, the additional cost incurred in paying for it is not significant. We do not include the VM provisioning delay here, as the number of VMs that can be afforded to launch will be determined by the amount of spare, or leftover, budget after this distribution.

Relying on the assumption that the more expensive the VM type the faster it is capable of processing tasks, the first step consists in finding the most expensive (or fastest) VM type (vmt_{ex}) where, if assigned to all tasks, their combined cost would be equal to or less than the budget. If no such type exist, then vmt_{ex} is defined to be the cheapest (or slowest) available VM type. If this is the case, although the estimated cost of running the workflow tasks on the cheapest VM type is higher than the budget, then we do not conclude the budget is insufficient to run the workflow, as at this stage we are overestimating the cost by assuming that VMs are not reused. This does, however, mean that there will be no spare budget to lease VMs, and the algorithm will be forced to re-use existing ones. Additionally, before accepting a budget plan that is higher than

ALGORITHM 1: Budget Distribution

```
1: procedure DISTRIBUTEBUDGET(β,T)
2:     levels = DAG levels
3:     Find fastest VM type vmt_ex such that IC = Σ_{i=1}^{|T|} C_{t_i}^{vmt_ex} ≤ β
4:     if No suitable IC ≤ β is found then
5:         vmt_ex = vmt_cheapest
6:     end if
7:     For every l ∈ levels do l.vmtype = vmt_ex
8:     For every t ∈ T do t.budget = C_t^{vmt_ex}
9:     if IC < β then
10:        spare = β − IC
11:        while spare > 0 and at least one level can be upgraded do
12:            for each level l ∈ levels with T_l ⊂ T tasks do
13:                vmt_up = next fastest vm than l.vmtype
14:                Previous level cost PLC = Σ_{i=1}^{|T_l|} C_{t_i}^{vmt_{l.vmtype}}
15:                New level cost NLC = Σ_{i=1}^{|T_l|} C_{t_i}^{vmt_up}
16:                if NLC − PLC ≤ spare then
17:                    l.vmtype = vmt_up
18:                    For every t ∈ T_l do t.budget = C_t^{vmt_up}
19:                    Update remaining spare budget
20:                end if
21:            end for
22:        end while
23:        if spare > 0 then
24:            for each level l ∈ levels do
25:                β_l = (|T_l|/|T|) ∗ spare
26:                l.provisioningBudget = β_l
27:            end for
28:        end if
29:    end if
30: end procedure
```

the actual budget, the algorithm checks that the available money is at least enough to run all of the remaining tasks in a single VM of the cheapest type, denoted as the minimum cost plan. Further details of this heuristic are explained in Section 4.4.

After determining vmt_{ex}, each task is assigned an initial budget corresponding to $C_t^{vmt_{ex}}$. BAGS then proceeds to distribute any spare or leftover budget by upgrading all tasks in a level using the following top-down strategy. Iteratively and starting at the top level of the DAG, all of the level's tasks are assigned additional budget corresponding to their execution on the next fastest VM type to vmt_{ex} if the total additional cost of running all the level's tasks on such VM type does not exceed the spare budget. This process is repeated until no more levels can be upgraded or the spare budget is exhausted.

Finally, any spare money left is distributed to each level for provisioning purposes. When a task is being scheduled, this provisioning budget will determine if a new VM can be launched or if an existing one has to be reused. The distribution is proportional to the number of tasks in the level. Algorithm 1 depicts an overview of the budget assignment process.

4.3. Resource Provisioning

This section explains the strategies used to create the resource provisioning plans for each of the BoT categories. A high-level overview of the process is depicted in Algorithm 2.

ALGORITHM 2: Resource Provisioning

```
1: procedure createResourceProvisioningPlan(bot)
2:     if bot ∈ BoT_hom then
3:         solve MILP for homogeneous bot
4:         for each vmt that had at least one task assigned do
5:             numTasks = number of tasks assigned to a VM of type vmt
6:             numVMs = number of VMs of type vmt used
7:             RP_vmt = (numTasks, numVMs)
8:             RP_bot ∪ RP_vmt
9:         end for
10:    else if bot ∈ BoT_het then
11:        solve MILP for heterogeneous bot
12:        for each vm that had at least one task assigned do
13:            tasks = tasks assigned to vm
14:            RP_vm = (tasks, vm)
15:            RP_bot ∪ RP_vm
16:        end for
17:    else if bot ∈ BoT_sin then
18:        t = bot.task
19:        vmt_fast = find fastest VM that can finish task t within bot.budget
20:        if vmt_fast does not exist then
21:            vmt_fast = vmt_cheapest
22:        end if
23:        RP_bot = (vmt_fast)
24:    end if
25: return RP_bot
26: end procedure
```

4.3.1. Bags of Homogenous Tasks. The resource capacity for bags of homogenous tasks is estimated using mixed integer linear programming (MILP). The MILP model was designed to provide an estimate of the number and types of VMs that can be afforded with the given budget so the tasks are processed with minimum makespan. The simplicity of the model was a main design goal, as a solution for large bags needs to be provided in a reasonable amount of time.

We recognize that although tasks are homogenous, their computation time may differ as the size of their input and output data may vary. For this reason, and to keep the MILP model simple, we assume all tasks in the bag take as long to process as the most data intensive task. That is, the task that uses and produces the most amount of data out of all the ones in the bag.

The following notation is used to represent some basic parameters used in the model:

—n: number of tasks in the bag,
—β: available budget to spend on the bag. The budget for a multi-task BoT is defined as the sum of the budgets of the individual tasks contained in the bag. If there is any spare budget assigned to the DAG level to which the tasks belong to, then this is added to the BoT budget as well,
—*IntTol*: refers to the MILP solver integrality tolerance. It specifies the amount by which an integer variable in the MILP can differ from an integer and still be considered feasible.

The following data sets representing the cloud resources are used as an input to the program:

—*VMT*: set of available VM types,
—VM_{vmt}: set of possible VM indexes for type vmt. Represents the number of VMs of the given type that can be potentially leased from the provider and ranges from 1 to n.

Each VM type is defined by the following characteristics:

—c_{vmt}: cost per billing period of VM type $vmt \in VMT$,
—$prov_{vmt}$: provisioning delay of a VM of type $vmt \in VMT$,
—$p_{t_{di}}^{vmt}$: processing time as calculated in Equation (3) of the most data intensive task $t_{di} \in BoT$ in VM type $vmt \in VMT$.

The following variables are used to solve the problem:

—M: makespan,
—$N_{vmt,k}$: integer variable representing the number of tasks assigned to the k^{th} VM ($k \in VM_{vmt}$) of type vmt,
—$L_{vmt,k}$: binary variable taking the value of 1 if and only if the k^{th} VM ($k \in VM_{vmt}$) of type vmt is to be leased, 0 otherwise,
—$P_{vmt,k}$: integer variable indicating the number of billing periods the k^{th} VM ($k \in VM_{vmt}$) of type vmt is used for.

The total number of time units the k^{th} VM ($k \in VM_{vmt}$) of type vmt is used for is defined as

$$U_{vmt,k}^{hom} = (p_{t_{di}}^{vmt} * N_{vmt,k}) + (prov_{vmt} * L_{vmt,k}) \tag{7}$$

and the total execution cost as

$$C_{botl} = \sum_{j \in VMT} \sum_{k \in VM_j} P_{vmt,k} * c_{vmt}. \tag{8}$$

The MILP is formulated as follows:

Minimize M Subject to:

$$\begin{aligned} M - U_{vmt,k} &\geq 0 \\ \forall\ vmt \in VMT, \quad \forall\ k \in VM_{vmt}, \end{aligned} \tag{C1}$$

$$\sum_{j \in VMT} \sum_{k \in VMT_j} N_{vmt,k} = n, \tag{C2}$$

$$\begin{aligned} N_{vmt,k} &\geq L_{vmt,k} \\ \forall\ vmt \in VMT, \quad \forall\ k \in VM_{vmt}, \end{aligned} \tag{C3}$$

$$\begin{aligned} N_{vmt,k} &\leq n * L_{vmt,k} \\ \forall\ vmt \in VMT, \quad \forall\ k \in VM_{vmt}, \end{aligned} \tag{C4}$$

$$\begin{aligned} U_{vmt,k}/\tau &\leq P_{vmt,k} \\ \forall\ vmt \in VMT, \quad \forall\ k \in VM_{vmt}, \end{aligned} \tag{C5}$$

$$\begin{aligned} (U_{vmt,k}/\tau) + (1 - IntTol) &\geq P_{vmt,k} \\ \forall\ vmt \in VMT, \quad \forall\ k \in VM_{vmt}, \end{aligned} \tag{C6}$$

$$C_{bot} \leq \beta. \tag{C7}$$

Constraint (C1) defines the BoT makespan as the longest time any of the leased VMs is used for. Constraint (C2) ensures all the tasks are processed. Constraints (C3) and

(C4) defines if a VM is leased based on the number of tasks assigned to it. Constraints (C5) and (C6) define the number of billing periods a VM is charged for by rounding up to the nearest integer the amount of time units the VM is used for. Finally, Constraint (C7) ensures the total cost does not exceed the budget.

After solving the problem, the variable $N_{vmt,k}$ is transformed into a resource provisioning plan of the form $RP^{hom}_{vmt} = (NumVms, NumTasks)$ for each VM type that has at least one task assigned to it. $NumVms$ indicates the number of VMs of type vmt to lease and $NumTasks$ the number of tasks that need to be processed by these VMs.

4.3.2. Bags of Heterogenous Tasks. The strategy used to plan the resource provisioning of heterogeneous tasks bags is also based on MILP. The model is similar to that of homogenous tasks. An additional set $T_{bot} = \{t_1, t_2, \ldots, t_n\}$ representing the tasks in the bag is included. The processing time of each task $t \in bot$ on each VM type vmt is represented by the parameter p^{vmt}_t. The binary variable $A_{t,vmt,k}$ is used to solve the problem in addition to M, $L_{vmt,k}$, and $P_{vmt,k}$. $A_{t,vmt,k}$ takes the value of 1 if and only if task t is allocated to the k_{th} VM of type vmt.

The total number of time units the k_{th} VM ($k \in VM_{vmt}$) of type VMT is used for is defined as

$$U^{het}_{vmt,k} = \sum_{t in T_{bot}} (p^{vmt}_t * A_{t,vmt,k}) + (prov_{vmt} * L_{vmt,k}), \tag{9}$$

and the total execution cost is defined by Equation (8).

The MILP is formulated in the same way as in Section 4.3.1 with the following differences:

—Constraint (C2) is reformulated to ensure that all the tasks are processed and that each task is assigned to a VM only once,

$$\sum_{j \in VMT} \sum_{k \in VMT_j} A_{t,vmt,k} = 1 \quad \forall\, t \in T_{bot}. \tag{C2}$$

—Constraints (C3) and (C4) are reformulated in terms of the variable $A_{t,vmt,k}$,

$$\sum_{t \in T_{bot}} A_{t,vmt,k} \geq L_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}, \tag{C3}$$

$$\sum_{t \in T_{bot}} A_{t,vmt,k} \leq n * L_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}. \tag{C4}$$

After solving the problem, the variable $A_{t,vmt,k}$ is transformed into a resource provisioning and scheduling plan of the form $RP^{het}_{vm} = (vm, T_{vm} \subset T_{bot})$ for each VM that had at least one task assigned to it. Notice that this provisioning plan determines the actual machines to use and the tasks that they are required to run, as opposed to just indicating the number and type of VMs to use. Due to the complexity of the MILP, heterogenous BoTs are limited in size to a constant N^{het}_{bot}. This constant is provided as a parameter to the algorithm and ensures the proposed MILP is solved in a reasonable amount of time. Bags larger than this parameter are split so they contain at most N^{het}_{bot} tasks.

4.3.3. Bags with a Single Task. This heuristic finds the fastest VM type that can be afforded with the budget assigned to the task. This is done by estimating the runtime of the task and its associated cost using Equation (6) on each available VM type. The one that can finish the task with minimum time and within the budget is selected and a resource provisioning plan of the form $RP^{sin} = (vmt_{fastest})$ is assigned to the task.

4.4. Scheduling

The scheduling is done by processing tasks that are in the scheduling queue and ready for execution. Each time the queue is processed, a max-min strategy is used, and tasks are sorted in ascending order based on their predicted runtime on the slowest VM type. In this way, we ensure that larger tasks from multi-task bags are scheduled first.

For each ready task, the first step is to identify the bag *bot* to which it belongs to. Afterwards, the algorithm determines if the bag *bot* has an active resource provisioning plan associated to it. If such plan has not been created yet, then the budget distribution is updated based on the remaining tasks and budget. A provisioning plan is then created considering the type of the bag, its budget, and the spare budget assigned to the corresponding DAG level. The latter value will determine the number of VMs that can be launched to process the bag. Once this plan is created, all the other tasks belonging to the bag (i.e., $\forall\ t_i \in bot$) will be scheduled based on it. In this way, the mathematical models only need to be solved once for each bag, when the first task of the bag is found in the scheduling queue.

Each active provisioning plan has a set of VMs, VM_{bot}, that were leased to serve the corresponding *bot*. This set is composed of busy VMs (VM_{bot}^{busy}) that are running tasks and idle VMs (VM_{bot}^{idle}) that can, and should, be reused by tasks in the bag. Once a VM is not required to process more tasks in the bag, it is removed from VM_{bot} and placed in a general-purpose VM set. This set, VM_{gp}^{idle}, contains idle VMs that can be reused by any task from any bag. VMs in this set that are approaching their next billing cycle are shut down to avoid incurring in additional billing periods.

Once *bot* has an associated resource provisioning plan RP_{bot}, then the task $t \in bot$ being processed can be scheduled. For bags with a single task, the algorithm first tries to reuse an existing VM from VM_{gp}^{idle}. The purpose is to avoid the cost and time overhead of provisioning delays, to reduce cost by using idle time slots, and to reduce the number of data transfers to the storage by assigning tasks to VMs that contain all or some of their input data. An idle VM is chosen if it can finish the task at least as fast as it was expected by its provisioning plan and with a cost less than or equal to its budget. If multiple free VMs fulfill these conditions, then the one that can finish the task the fastest is selected. In this way, tasks are encouraged to run on the same resources as their parent tasks, as they are expected to have smaller runtimes in VMs where their input data are readily available. If no idle VM is found, then a new one of the type specified by the plan is leased if the level's spare budget allows for it. If not, then the task is put back in the queue to be scheduled later on an existing VM that becomes available.

Tasks belonging to bags of homogenous tasks are processed in a similar way. The first step is to try to map the task to a free VM in VM_{bot}^{idle}. VMs in this set are sorted in ascending CPU capacity order; in this way, the most powerful VM is always reused first. This in conjunction with the max-min strategy used to sort tasks ensures that the largest tasks get assigned to the fastest VMs when possible. If there are no VMs in VM_{bot}^{idle}, then the algorithm tries to schedule the task on a free VM from VM_{gp}^{idle} that can finish the task for the same or a cheaper price than the most expensive VM type in the provisioning plan. If no suitable idle VM is found, then the provisioning

ALGORITHM 3: Scheduling

```
1: procedure SCHEDULEQUEUE(Q)
2:     while Q is not empty do
3:         t = Q.poll()
4:         bot = getBoT(t)
5:         if no provisioning plan RP exists for bot then
6:             β_r = remaining budget
7:             T_r = remaining tasks
8:             distributeBudget(β_r, T_r)
9:             update bot budget
10:            update t.level spare budget
11:            RP_bot = createResourceProvisioningPlan(bot)
12:        end if
13:        if there is an idle VM vm_bot^idle ∈ VM_bot^idle then
14:            schedule(t, vm_bot^free)
15:        else if there is a suitable general purpose idle VM vm_gp^idle ∈ VM_gp^idle then
16:            schedule(t, vm_gp^idle)
17:        else
18:            if bot ∈ BoT_hom and bot.hasRemainingVMQuota() then
19:                vmType = RP_bot.nextVMTypeToLease()
20:                vm^new = provisionVM(vmType)
21:                schedule(t, vm^new)
22:            else if bot ∈ BoT_het then
23:                vm = RP_bot.getVmForTask(t)
24:                if vm is not leased then
25:                    vm = provisionVM(vm.vmType)
26:                end if
27:                schedule(t, vm)
28:            else if bot ∈ BoT_sin and t.level spare budget is enough to lease RP_bot.VMType then
29:                vm^new = provisionVM(RP_bot.VMType)
30:                schedule(t, vm^new)
31:            else
32:                place t back in queue
33:            end if
34:        end if
35:    end while
36: end procedure
```

plan is executed in the following way. If the number of VMs currently leased for the provisioning plan is less than the specified one, then a new VM can be leased to run the task. The fastest VM type of those still available is chosen. If the VM quota has been reached and all the necessary VMs have been leased, then the task is put back in the queue so it can be mapped to an existing VM assigned to the *bot* provisioning plan during the next scheduling cycle.

Tasks from heterogeneous bags are simply scheduled onto the VM specified by their provisioning plan. If the VM has already been leased, then the task is added to the queue of jobs waiting to be processed by the VM. If it has not been leased, then it is provisioned and the task assigned to it.

Finally, we define the minimum cost required to run a set of tasks T as the cost of running all the tasks sequentially on a single VM of the cheapest type,

$$C_T^{min} = \left\lceil \left(\sum_{i=1}^{|T|} P_{t_i}^{vmt_{cheapest}} \right) \Big/ \tau \right\rceil * c_{vmt_{cheapest}}. \tag{10}$$

Whenever $C_T^{min} > \beta$ or there is no feasible solution to a MILP problem, then the minimum cost plan is put in place. This plan consists on assigning every task to a

Table I. VM Types Based on Google Compute Engine Offerings

Name	Memory	Google Compute Engine Units	Price per Minute
n1-standard-1	3.75GB	2.75	$0.00105
n1-standard-2	7.5GB	5.50	$0.0021
n1-standard-4	15GB	11	$0.0042
n1-standard-8	30GB	22	$0.0084

single VM of the cheapest available type. This is done with the aim of reducing cost as much as possible until the algorithm recovers or to finish the execution of the workflow with a cost as close to the budget as possible.

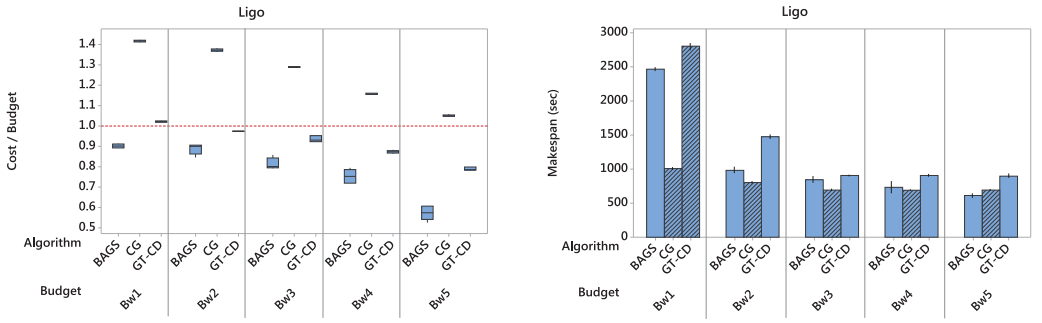
5. PERFORMANCE EVALUATION AND RESULTS

The performance of BAGS was evaluated using five well-known workflows from different scientific areas, each with approximately 1,000 tasks. The Montage application from the astronomy field is used to generate custom mosaics of the sky based on a set of input images. Most of its tasks are characterized by being I/O intensive while not requiring much CPU processing capacity. The Ligo workflow from the astrophysics domain is used to detect gravitational waves. It is composed mostly of CPU intensive tasks with high memory requirements. SIPHT is used in bioinformatics to automate search for sRNA encoding-genes. Most of the tasks in this workflow have high CPU and low I/O utilization. Also in the bioinformatics domain, the Epigenomics workflow is a CPU intensive application that automates the execution of various genome-sequencing operations. Finally, CyberShake is used to characterize earthquake hazards by generating synthetic seismograms and can be classified as a data-intensive workflow with large memory and CPU requirements. The workflows are depicted in Figure 1, and their full description and characterization is presented by Juve et al. [2013].

An IaaS provider offering a single data center and four types of VMs was modeled using CloudSim [Calheiros et al. 2011]. The VM configurations are based on those offered by Google Compute Engine and are shown in Table I. A VM billing period of 60s was used. For all VM types, the provisioning delay was set to 60s. CPU performance variation was modeled after the findings by Schad et al. [2010]. The performance of a VM was degraded by at most 24% based on a normal distribution with a 12% mean and a 10% standard deviation. Based on the same study, the bandwidth available for each data transfer within the data center was subject to a degradation of at most 19% based on a normal distribution with a mean of 9.5% and a standard deviation of 5%. The described CPU degradation configuration was used in all of the experiments except those in Section 5.3 while the specified data transfer degradation was used throughout all of the experiment sets. A global shared storage with a maximum reading and writing speeds was also modeled. The reading speed achievable by a given transfer is determined by the number of processes currently reading from the storage, and the same rule applies for the writing speed. In this way, we simulate congestion when trying to access the storage system.

The experiments were conducted using five different budgets, with β_{W1} being the strictest one and β_{W5} being the most relaxed one. For each workflow, β_{W1} is equal to the cost of running all the tasks in a single VM of the cheapest type. β_{W5} is the cost of running each workflow task on a different VM of the most expensive type available. An interval size of $\beta_{int} = \beta_{W5} - \beta_{W1}/4$ is then defined and used to estimate the remaining budgets: $\beta_{W2} = \beta_{W1} + \beta_{int}$, $\beta_{W3} = \beta_{W2} + \beta_{int}$, and $\beta_{W4} = \beta_{W3} + \beta_{int}$.

Two algorithms were used when evaluating the performance of BAGS. The first one is called GreedyTime-CD [Yu et al. 2009] (GT-CD) and was developed for utility grids. It distributes the budget to tasks based on their average execution times. At



(a) (b)

Fig. 2. Makespan and cost experiment results for the Ligo workflow.

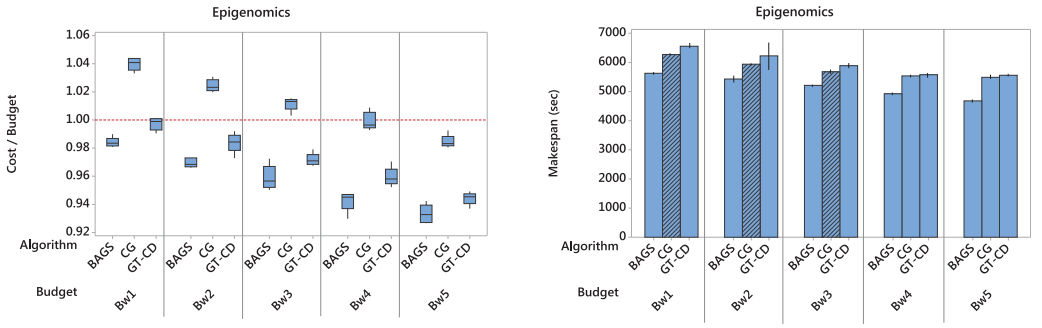
runtime, VMs that can finish the tasks with minimum time within their budget are selected. We adapted GT-CD to dynamically lease VMs based on a task's assigned budget. This auto-scaling mechanism was designed so leased VMs are reused when possible without impacting the original schedule produced by the algorithm. The second one is the Critical-Greedy [Wu et al. 2015] (CG) budget-constrained algorithm. It was developed for IaaS cloud environments and makes an initial estimate of cost boundaries for each task based on the available budget and VM types. Any additional budget is distributed to each task based on a time and cost difference ratio. CG ignores billing periods when calculating the cost of using a VM type, and it does not specify how to allocate tasks to actual VMs. We adapted the algorithm to consider billing periods when estimating the task's cost boundaries and introduced the same VM auto-scaling mechanism implemented for GT-CD.

5.1. Algorithm Performance

The goal of these experiments is to evaluate the performance of the algorithms in terms of cost and makespan. The cost performance is determined by an algorithm's ability to meet the specified budget constraint, this is evaluated by using the workflow's cost to budget ratio. In this way, ratio values greater than one indicate a cost larger than the budget, values equal to one a cost equal to the budget, and values smaller than one a cost smaller than the budget. The experiments for each budget interval, workflow, and algorithm were repeated 20 times. The box plots displaying the cost to budget ratios summarize these data while the bar charts depicting the workflow's makespan show the mean value obtained from the data and the 95% confidence interval for the mean. The dashed bars in the makespan bar charts indicate that the mean cost obtained by the algorithm exceeded the corresponding budget.

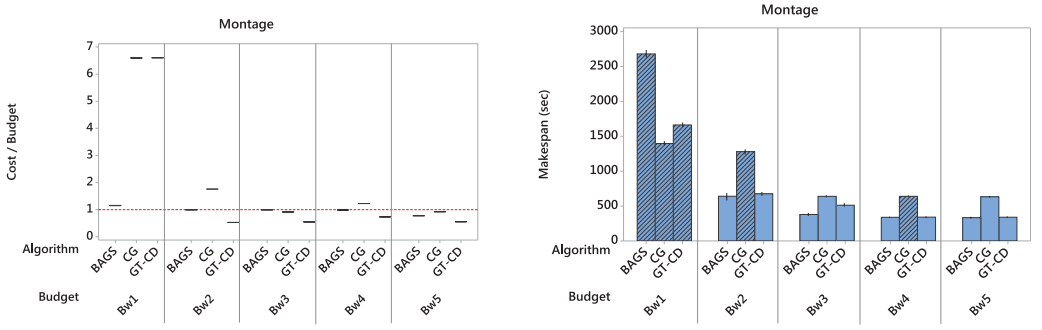
The results obtained for the Ligo workflow are shown in Figure 2. BAGS is the only algorithm capable of achieving a ratio smaller than one for all of the five budget intervals. The mean ratio obtained by GT-CD is below one from the second to the fifth budget intervals, while CG fails to meet the budget in all of the five cases. In every scenario in which BAGS and GT-CD meet the budget, BAGS achieves a lower makespan, demonstrating its ability to generate high-quality schedules.

Figure 3 depicts the results obtained for the Epigenomics application. Both BAGS and GT-CD are successful in meeting the five budget constraints, while CG meets the last three. BAGS always achieves the lowest makespan of those algorithms that complete the execution within budget. These results demonstrate once again the efficiency of the makespan-minimizing heuristics used in BAGS.



(a) (b)

Fig. 3. Makespan and cost experiment results for the Epigenomics workflow.



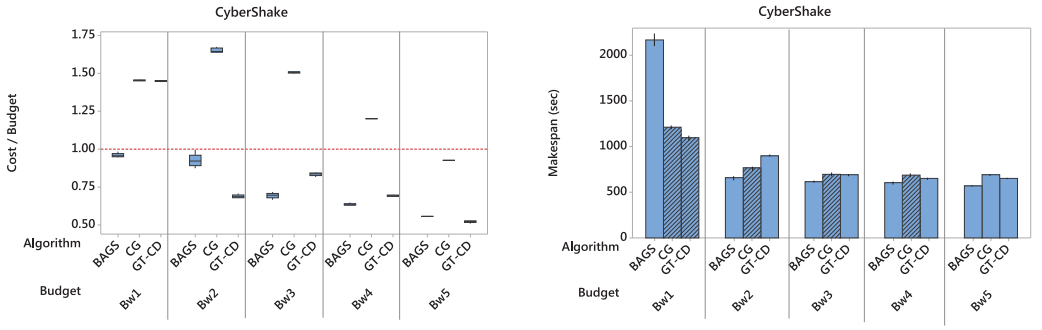
(a) (b)

Fig. 4. Makespan and cost experiment results for the Montage workflow.

The results for the Montage application are shown in Figure 4. The first budget constraint proves too tight for any of the algorithms to meet it. However, the ratio obtained by BAGS is considerably smaller than the ratio obtained by the other algorithms. For the second and third budget intervals, BAGS outperforms those algorithms capable of meeting the budget by obtaining lower makespans. The fourth budget interval sees GT-CD and BAGS obtain very similar average makespans, and in this case, GT-CD obtains a lower ratio when compared to BAGS. All of the algorithms are successful in meeting the final budget interval, with BAGS and GT-CD obtaining once again very similar makespans that are considerably smaller than the ones obtained by CG.

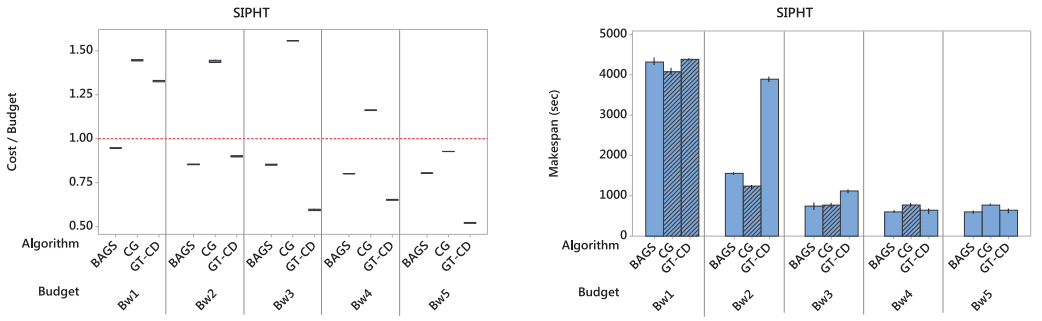
The CyberShake workflow results are shown in Figure 5. The first budget constraint is too strict for either GT-CD or CG to meet it. BAGS demonstrates its ability to deal with unexpected delays by being the only algorithm capable of staying within this budget. For the rest of the budget intervals, BAGS outperforms in every case the other algorithms in terms of makespan. In the cases of β_{w3} and β_{w4} BAGS not only achieves the fastest time but also the cheapest cost.

Figure 6 shows the results obtained for the SIPHT workflow. BAGS succeeds in meeting the budget in every case, GT-CD meets the four most relaxed constraints, and CG meets only the last budget interval. In all of the five scenarios, BAGS outperforms the other algorithms by generating lower makespan schedules.



(a) (b)

Fig. 5. Makespan and cost experiment results for the CyberShake workflow.



(a) (b)

Fig. 6. Makespan and cost experiment results for the SIPHT workflow.

Overall, BAGS is the most successful algorithm in meeting the budget constraints by achieving its goal in all of the scenarios except one, the first budget interval of the Montage workflow. Even in this case, it performs better than the other algorithms by having a ratio value approximately 6 times smaller than that of GT-CD and CG. This demonstrates the importance of tailoring an algorithm to consider the underlying cloud characteristics to take advantages of the features offered by the platform and meet the Quality of Service (QoS) requirements. The experiments also demonstrate the efficiency of BAGS in generating higher-quality schedules by achieving a lower makespan values in every case except one (Montage workflow, β_{W4}). These results highlight the efficiency of the time optimization strategies used by BAGS. Another desirable characteristic of BAGS that can be observed from the results is its ability to consistently decrease the time it takes to run the workflow as the budget increases. The importance of this relies in the fact that many users are willing to trade off execution time for lower costs while others are willing to pay higher costs for faster executions. The algorithm needs to behave within this logic in order for the budget value given by users to be meaningful.

5.2. Provisioning Delay Sensitivity

Fine-grained billing periods encourage frequent VM provisioning operations and therefore, it is important to evaluate the ability of BAGS to finish the workflow execution with a cost no greater than the given budget under different VM provisioning delays. The delays were varied from zero to nine billing periods (540s). Figure 7 shows the

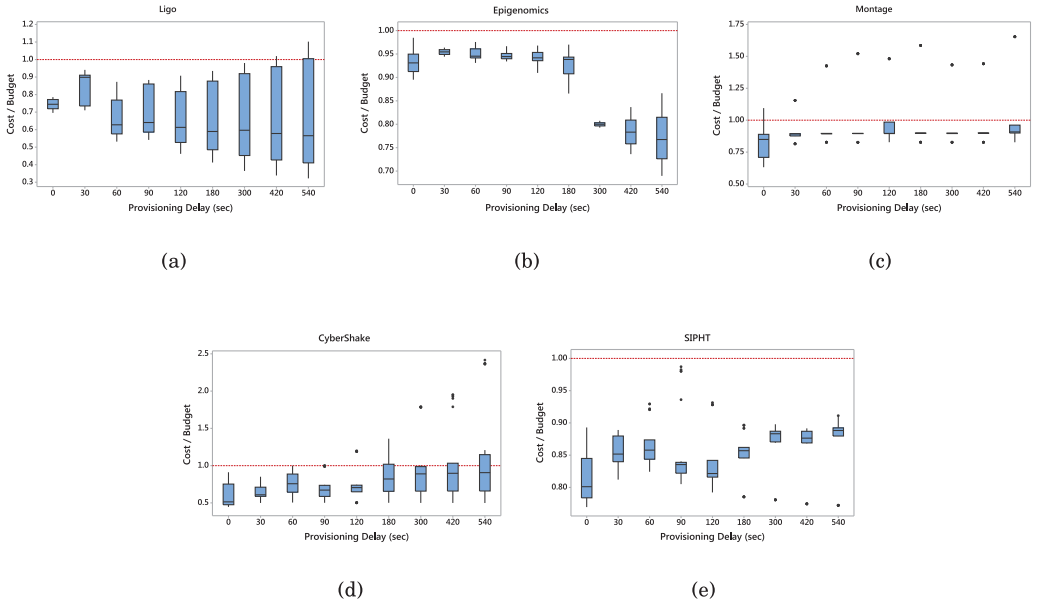


Fig. 7. Cost to budget ratios obtained for each of the workflows with varying VM provisioning delays.

ratios of cost to budget obtained for the each of the workflow applications across all five budgets. Identical outlier data points are displayed as a single symbol.

For the Ligo application, the mean and median ratio values remained under one for all of the provisioning delays. However, for the last two values, 420s and 540s, the maximum ratio value obtained is slightly higher than one. This is due to the algorithm being unable to meet the first budget interval, as it becomes too strict for it to be achievable with such high provisioning delays. The results for the Montage workflow display maximum or outlier values greater than one in every case, this is inline with what was found when analyzing the performance of the algorithms, the first budget is too strict for BAGS to finish on time regardless of the provisioning delay. The mean and median values, however, remain well below one in every case. For the CyberShake application, outliers greater than one start to appear from a provisioning delay value of 120s onwards. Once again, these ratios correspond to the strictest budget and they increase in value as the delay increases. In the Epigenomics and SIPHT cases, all of the ratio data points are below one, demonstrating the ability of BAGS to adapt to increasing provisioning delays as long as the budget allows for it.

5.3. Performance Degradation Sensitivity

Recognising performance variability is important for schedulers so they can recover from unexpected delays and fulfill the QoS requirements. The sensitivity of the algorithm to VM CPU performance variation was studied by analyzing the cost to budget ratio under different degradation values. It was modeled using a normal distribution with a variance of 1% and different average and maximum values. The average values were defined as half of the maximum CPU performance degradation which range from 0% to 80%.

The results obtained are depicted in Figure 8, identical outlier data points are displayed as a single symbol. The mean and median ratio values are under one for all of the degradation values for the Ligo application. With an 80% maximum degradation, however, the maximum ratio obtained is just over one and corresponds to the strictest

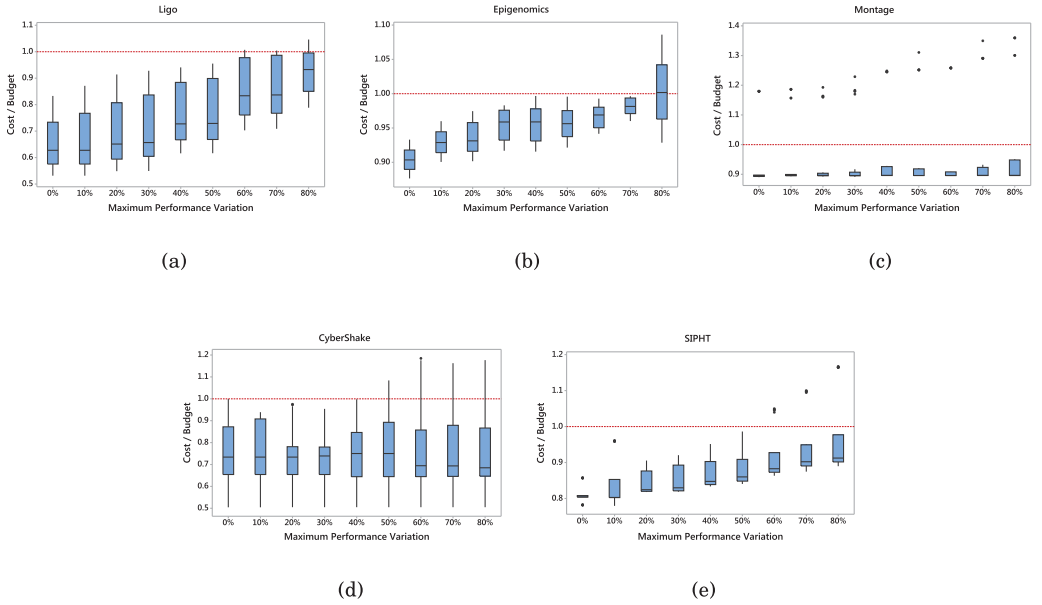


Fig. 8. Cost to budget ratios obtained for each of the workflows with different CPU performance variation values.

budget value. The results are similar for the Epigenomics application, but in this case, a greater sensitivity to the unexpected delays is seen in the case of 80% maximum degradation, with the median being slightly higher than one. The outliers displayed in the Montage box plot correspond once again to the first budget, which is too strict to be met regardless of the provisioning delay or performance variation. All the other ratios obtained remained under one for this application. The CyberShake workflow is more sensitive to degradation with the maximum ratio values exceeding one from 50% onwards. These belong to the strictest budget as the mean and median values are well below one in all of the cases. Finally, the results obtained for SIPHT demonstrate the algorithm is capable of finishing within budget in most of the cases, except for some outlier data points for the three greater performance variation values.

Another potential cause for exceeding the budget constraint is the fact that BAGS creates a static provisioning plan for BoTs with multiple tasks. Although this enables the algorithm to make better optimization decisions to minimize the makespan of workflows, it also affects its responsiveness to changes in the environment. These results demonstrate, however, that despite this, BAGS is still successful in achieving its budget goal in the vast majority of cases. As a future work, a rescheduling strategy for multi-task BoTs will be explored with the aim of further reducing the impact of unexpected delays.

5.4. Mathematical Models Solve Time

The time taken to solve the MILP models for homogeneous and heterogeneous bags was also studied. The number of tasks used as input to the homogeneous BoT model was varied from 10 to 1,000 while the number of tasks for the heterogeneous BoT model was varied from 10 to 100. For each of these values, experiments using 10 different budget values ranging from stricter to more relaxed ones were performed. Figures 9 and 10 summarize the results obtained. Both models were formulated in AMPL and solved using the default configuration of CPLEX.

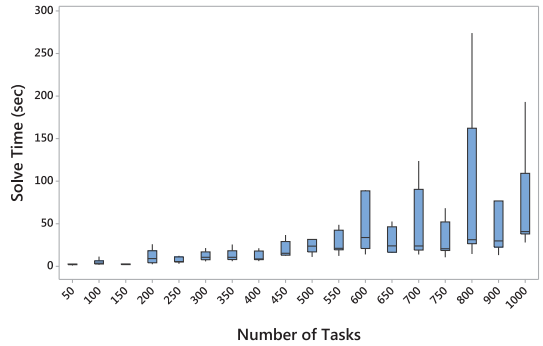


Fig. 9. Solve time for the homogeneous BoT MILP model. The results display the time for solving the MILP with four different VM types and across 10 different budgets, ranging from stricter to more relaxed ones.

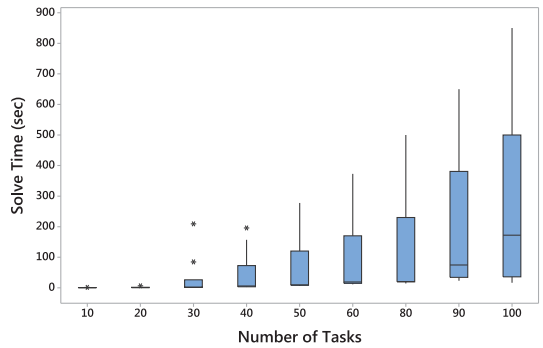


Fig. 10. Solve time for the heterogeneous BoT MILP model. The results display the time for solving the MILP with four different VM types and across 10 different budgets, ranging from stricter to more relaxed ones.

The results obtained for the homogeneous BoT case demonstrate the scalability of the proposed model with the maximum time taken to solve the problem being approximately 4.5min for 800 tasks and the largest median value being 40s for 1,000 tasks. The performance of the heterogeneous BoT model, however, is greatly affected by the number of tasks being scheduled. For 100 tasks, the maximum solve time obtained is in the order of 14min, this value is too high and unpractical for our scheduling scenario. Based on these results, the maximum number of tasks, N_{bot}^{het}, allowed in an heterogeneous bag was defined as 50, for which we obtained a maximum solve time of approximately 4.5min.

6. CONCLUSIONS

BAGS, an adaptive resource provisioning and scheduling algorithm for scientific workflows in clouds capable of generating high-quality schedules, was presented in this article. It has as objective minimizing the overall workflow makespan while meeting a user-defined budget constraint. The algorithm is dynamic to respond to unexpected delays and environmental dynamics common in cloud computing. It also has a static component to schedule groups of tasks that allows it to find the optimal schedule for a set of workflow tasks improving the quality of the schedules it generates.

The simulation experiments show that our solution has an overall better performance than other state-of-the-art algorithms. It is successful in meeting the strictest budgets under unpredictable situations involving CPU and network performance

variation as well as VM provisioning delays. As a future work, a rescheduling strategy for multi-task BoTs will be explored with the aim of further reducing the impact of performance degradation. Different budget distribution strategies as well as a more scalable heuristic to schedule heterogeneous BoTs will also be studied.

REFERENCES

Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, and Seungryoul Maeng. 2011. Cost optimized provisioning of elastic resources for application workflows. *Future Gener. Comput. Syst.* 27, 8 (2011), 1011–1026.

Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. 2011. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.* 41, 1 (2011), 23–50.

Thiago A. L. Genez, Luiz F. Bittencourt, and Edmundo R. M. Madeira. 2012. Workflow scheduling for SaaS/PaaS cloud providers considering two SLA levels. In *Proceedings of the Network Operations Management Symposium (NOMS'12)*.

Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. 2007. Examining the challenges of scientific workflows. *IEEE Comput.* 40, 12 (2007), 26–34.

Google. 2015a. Amazon Simple Storage Service. (Nov 2015). Retrieved October 2016 from http://aws.amazon.com/s3/.

Google. 2015b. Google Cloud Storage. (Nov 2015). Retrieved October 2016 from https://cloud.google.com/storage/.

Google. 2015c. Google Compute Engine. (Nov 2015). Retrieved October 2016 from https://cloud.google.com/compute/.

A. Gupta and D. Milojicic. 2011. Evaluation of HPC applications on cloud. In *Proceedings of the 2011 6th Open Cirrus Summit (OCS'11)*. 22–26.

A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst.* 22, 6 (June 2011), 931–945.

Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. 2010. Performance analysis of high performance computing applications on the amazon web services cloud. In *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom)*.

Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. 2013. Characterizing and profiling scientific workflows. *Future Gener. Comput. Syst.* 29, 3 (2013), 682–692.

Maciej Malawski, Kamil Figiela, Marian Bubak, Ewa Deelman, and Jarek Nabrzyski. 2015. Scheduling multilevel deadline-constrained scientific workflows on clouds based on cost optimization. *Sci. Program.* 2015, 5 (2015).

Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. 2012. Cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, and Storage Analysis (SC'12)*.

Ming Mao and Marty Humphrey. 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of the International Conference on High Performance Computing, Networking, and Storage Analysis (SC'11)*.

Ming Mao and Marty Humphrey. 2013. Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS'13)*. IEEE, 67–78.

Microsoft. 2015. Microsoft Azure. (Nov 2015). Retrieved October 2016 from https://azure.microsoft.com.

Simon Ostermann, Alexandria Losup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. 2010. A performance analysis of EC2 cloud computing services for scientific computing. In *Cloud Computing*. Springer, 115–131.

Ilia Pietri, Maciej Malawski, Gideon Juve, Ewa Deelman, Jarek Nabrzyski, and Rizos Sakellariou. 2013. Energy-constrained provisioning for scientific workflow ensembles. In *Proceedings of the International Conference on Cloud Green Computing (CGC'13)*.

Deepak Poola, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2014. Fault-tolerant workflow scheduling using spot instances on clouds. *Proc. Comput. Sci.* 29 (2014), 523–533.

Rackspace. 2015. Rackspace Block Storage. (Nov 2015). Retrieved October 2016 from http://www.rackspace.com.au/cloud/block-storage.

Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.* 3, 1–2 (2010), 460–471.

Jianwu Wang, Prakashan Korambath, Ilkay Altintas, Jim Davis, and Daniel Crawl. 2014. Workflow as a service in the cloud: Architecture and scheduling algorithms. *Proc. Comput. Sci.* 29 (2014), 546–556.

Chunlin Wu, Xingqin Lin, Daren Yu, Wei Xu, and Luoqing Li. 2015. End-to-end delay minimization for scientific workflows in clouds under budget constraint. *IEEE Trans. Cloud Comput*. 3, 2 (2015), 169–181.

Zhangjun Wu, Zhiwei Ni, Lichuan Gu, and Xiao Liu. 2010. A revised discrete particle swarm optimization for cloud workflow scheduling. In *Proceedings of the International Conference on Computational Intelligence Security (CIS'10)*.

Jia Yu, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2009. Deadline/budget-based scheduling of workflows on utility grids. *Market-Oriented Grid and Utility Computing* (2009), John Wiley & Sons, Inc., 427–450.

Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. 2012. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *Proceedings of the International Conference on Advanced Information Network Applications (AINA'12)*.

Amelie Chi Zhou, Bingsheng He, and Cheng Liu. 2016. Monetary cost optimizations for hosting workflow-as-a-service in IaaS clouds. *IEEE Trans. Cloud Comput.* 4, 1 (2016), 34–48.

Received December 2015; revised October 2016; accepted January 2017