Budget-aware workflow scheduling with DIET

Aurélie Kong Win Chang, Yves Caniou, Eddy Caron, Yves Robert

Budget-aware workflow scheduling with DIET

Yves Caniou, Eddy Caron, Aurélie Kong Win Chang, Yves Robert

RESEARCH REPORT
N° 9381
December 2020
Project-Team AVALON & ROMA

ISSN 0249-6399   ISRN INRIA/RR--9381--FR+ENG

Budget-aware workflow scheduling with DIET

Yves Caniou[*], Eddy Caron[*], Aurélie Kong Win Chang[*], Yves Robert[*][†]

Project-Team AVALON & ROMA

Research Report n° 9381 — December 2020 — 26 pages

Abstract: DIET is a widely used workflow engine capable of scheduling workflows dynamically. This feature enables the engine to react to various events occurring during execution. However, a static schedule calculated before the submission of the workflow can provide many optimization opportunities, in particular when budget constraints should be enforced. In this paper, we expand DIET by adding a static scheduling functionality. To validate this new functionality on a realistic test case, we compare the behavior of budget-aware algorithms designed in [5], both in simulation and on an actual platform, Grid'5000. The obtained results are very similar, showing that the benefits provided by the scheduling algorithms are effective, both in simulation and in a real execution.

Key-words: workflow, scheduling, cloud, budget-aware, DIET.

[*] LIP, École Normale Supérieure de Lyon, CNRS & Inria, France
[†] University Tennessee Knoxville, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes d'ordonnancement de workflows avec contrainte de budget dans DIET

Résumé : DIET est un moteur de workflow très utilisé, capable d'ordonnancer des workflows de manière dynamique. Cette fonctionnalité lui permet de réagir aux divers événements survenant au cours de l'exécution des workflows. Cependant, un ordonnancement statique calculé avant la soumission du workflow peut offrir de nombreuses opportunités d'optimisation, en particulier dans le cas d'une contrainte de budget. Dans ce rapport, nous étendons les fonctionnalités de DIET en y ajoutant la capacité de faire des ordonnancements statiques. Pour valider ce nouveau module avec un cas d'étude réaliste, nous comparons le comportement d'algorithmes présentés dans [5], à la fois par des simulations et sur une vraie plateforme, Grid'5000. Les résultats obtenus sont très similaires, montrant que les avantages apportés par les algorithmes d'ordonnancement sont réels, que ce soit dans le cadre d'une simulation ou d'une vraie exécution.

Mots-clés : workflow, ordonnancement, contrainte de budget, DIET.

1 Introduction

Public Cloud has emerged as an interesting tool for scientists, offering an infrastructure adaptable on demand, with a variety of available options and performances. Multiple workflow engines for Cloud have been designed, with more and more functionalities [8, 9, 10], in order to provide users with the best experience. These workflow engines aim at helping users to pick the most appropriate resources for their intended work, in terms of the number and characteristics of the Virtual Machines (VMs) selected for execution. The main objective is to enable easy-to-produce applications while guaranteeing that, given a constrained budget, rented resources are used up to the maximum of their capacity.

In this paper, we focus on the DIET workflow engine [7, 8, 6]. Among other functionalities, this middleware offers an efficient dynamic workflow scheduling. However, a static schedule calculated before the submission of the workflow can provide many optimization opportunities, in particular when budget constraints should be enforced. We demonstrated such opportunities through a comprehensive simulation campaign in our previous work [5], where we compared the behavior of several budget-aware algorithms. One major objective of this work is to further validate the conclusions of the simulation study by conducting real-life experiments, with the same scientific workflows, on the French national validation platform Grid'5000. This requires to expand DIET by adding a static scheduling functionality, and to validate its design and performance by comparing the results of the simulations with the actual executions on Grid'5000. Adding a static scheduling functionality to DIET requires to design and implement major extensions to the engine, and we describe the new features in detail in the following sections.

The main contributions of this paper are twofold. First, by running real-life experiments, we obtain a proof of concept for several static budget-aware scheduling algorithms [5], assessing the accuracy of simulation results on a real-life platform. We also provide an important new functionality within DIET, along with a tool that allows users to provide static schedules of their own, and then automatically generates DIET code to execute the target workflows. Thus the DIET workflow engine can now work efficiently in two ways: as a dynamic multi-workflow scheduler (the default mode for this middleware), and as a static scheduler with the benefit of highly-tuned algorithms whenever more information on task weights and file sizes are available.

The rest of the paper is organized as follows. We first study related work in Section 2. Then, in Section 3, we introduce the DIET middleware and describe the new DIET functionality. In Section 4, we overview the budget-aware algorithms that we aim at comparing. Section 5 details the experimental framework. Results are reported in Section 6, with a comparison analysis between real executions on Grid'5000 and the corresponding simulations.

2 Related work

We briefly discuss related work in this section. Section 2.1 deals with workflow engines, while Section 2.2 provides references on budget-aware scheduling algorithms.

2.1 Workflow engines

Many scientific applications from various disciplines are structured as workflows [3]. Informally, a workflow can be seen as the composition of a set of basic operations that have to be performed on a given input data set to produce the expected scientific result. For a long time, the development of complex middleware with workflow engines [8, 9, 10] has automated workflow management. For example, the Pegasus Workflow Management System[1] [10] maps workflows on resources until a given horizon beyond which it considers pre-scheduling is inefficient, and allows its users to plug their own scheduling algorithms if needed. Steep[2] [15] comes along its own way to schedule workflows, submitting complete process chains to its remote agents, and supports cyclic workflows graphs. Apache Airflow[3] [2] is more oriented on accessibility to most users, hence its focus on the user interface and the use of Python to describe workflows or interact with the engine. In [19], the authors summarize the key features of four production-ready WMSs: Pegasus, Makeflow, Apache Airflow, and Pachyderm. For this work, we focus on DIET[4] [6] because of its practicality, and the possibility to extend it with additional modules.

Infrastructure as a Service (IaaS) Clouds raised a lot of interest recently thanks to an elastic resource allocation and pay-as-you-go billing model. A Cloud user can adapt the execution environment to the needs of his application on a virtually infinite supply of resources. While the elasticity provided

[1] `https://pegasus.isi.edu/`

[2] `https://steep-wms.github.io/`

[3] `https://airflow.apache.org/`

[4] `https://graal.ens-lyon.fr/~diet/`

by IaaS Clouds gives way to more dynamic application models, it also raises new issues from a scheduling point of view. An execution now corresponds to a certain budget, which imposes some constraints on the scheduling process. In [12], the authors propose a performance-feedback autoscaler that is budget-aware: using Apache Airflow, they tackle the same scheduling problem as in this paper, but they focus on the auto-scaling problem (allocating and de-allocating resources on the fly).

2.2 Algorithms

Scheduling scientific workflows in cloud is a well-studied domain [16, 23]. To the best of our knowledge, the closest papers to the budget-aware algorithms with stochastic execution times that we designed in [5] and compare in this paper, are [1] and [25], which both propose workflow scheduling algorithms (BDT in [1], CG/CG+ in [25]) under budget constraints, but with a simplified platform model. As in our previous work [21], we have extended BDT and CG/CG+ to enable a fair comparison with our algorithms. More recent scheduling algorithms exist, but address different aspects of the problem. For example, [26] aim at simultaneously minimizing the cost and makespan of workflow executions, but they use a simplified platform model without Cloud storage. Another study [18] uses a platform closer to ours, but with a completely different objective, namely the minimization of data transfers (which should eventually reduce both makespan and cost too). Finally, the workflows considered in [17] present an uncertainty in task durations, but the authors schedule multiple workflows at the same time instead of optimizing for a single workflow.

3 Framework

Scientific workflows are represented with a DAG (Directed Acyclic Graph) $G = (V, E)$, where V is the set of tasks to schedule and E is the set of dependencies between tasks. In this model, a dependency corresponds to a data transfer between two tasks. Workflows are scheduled on an heterogeneous platform representing an IaaS (Infrastructure as a Service) Cloud, with a tarification depending on the performances of the used machines, on the amount of data transferred to and out of the Cloud, and on the amount of time during which each machine has been used (see Section 4.2 for cost instances). The constraints are a limited budget and the objective is to minimize total execution time. In such an environment, it can be useful to statically study the structure of the workflow before its execution in order to make the best use as possible of our resources. This section details the improvements made to a kernel component of DIET, the workflow Master Agent, and all the tools that we have designed to enable such a static study and to fully automate the generation of the DIET elements to ease the user experience in executing real scientific workflows.

We start with an overview of the DIET workflow engine in Section 3.1. Next we describe the extended MA$_{DAG}$ for static scheduling in Section 3.2. Finally we deal with experiment-oriented tools in Section 3.3.

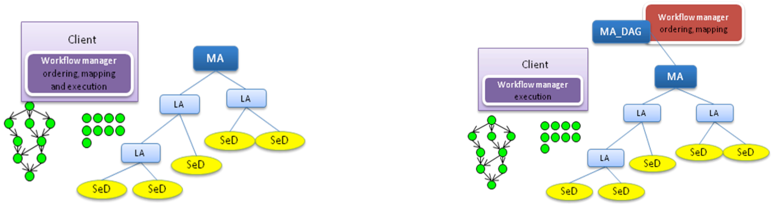
3.1 DIET workflow engine

DIET is a well established workflow engine [6, 7, 8]. A DIET platform consists of a hierarchy of agents scheduling the requests addressed by a client and sending them to the appropriate servers. DIET users, following the GridRPC paradigm, usually submit individual tasks. Workflows can of course be decomposed into individual tasks, but the knowledge of their overall graph structure helps the scheduler to make efficient mapping decisions. We have extended the agent hierarchy by adding a new special agent to handle workflow submissions. This special agent, called MA$_{DAG}$, manages the different workflow submissions. An overview of the DIET architecture is shown in Figure 1.

In the case of the execution of a workflow, the client sends the XML file describing the workflow to a MA$_{DAG}$ agent. This MA$_{DAG}$ agent will parse the description file, manage task dependencies and send the requests of ready tasks to the master agent from which it depends. The master agent will then determine which server can execute the requests, and send them to the appropriate available servers. Once the request received, the server will look for the files needed for execution, download them if they are not already on the server, and then execute the request.

The workflow support in DIET includes three modes. These three modes are represented by two architectures (Figure 1a and 1b) which are described below, and can be used in the same platform without conflict.

- The first one provides a workflow manager at the client side. The client reads and processes the workflow description; a Dag structure is created. The client sends a set of problem descriptions to the MA to check if all services are available. If all services are available, the client starts the workflow execution. In this mode, since all scheduling operations are done in the client side, we can use a customized scheduler. The client programmer can write his personal scheduler.



(a) Architecture 1: Complete schedule from the Client.

(b) Architecture 2: Schedule is handled by the MA_{DAG}

Figure 1: Two different architectures of the DIET workflow engine.

- The second one uses a special agent called MA_{DAG}, which is responsible to communicate with the MA of the platform. In this mode, DIET provides an ordering and mapping for workflow execution.

- The third mode is similar to the second one, but in this mode, the MA_{DAG} only provides an ordering for the workflow execution.

The use of the MA_{DAG} is based on the user's choice to use his own scheduling strategy or to use the global one provided by the MA_{DAG}. It is obvious that when the user decides not to use the MA_{DAG}, there is no collaboration between the different clients, but he can use and easily test a new scheduling algorithm by plugging it in the client code. On the other hand, when the MA_{DAG} is used, the workflow submissions go through this special agent and the multi-workflows can be handled more efficiently using core heuristics. To avoid overloading due to multiple workflow submissions from different clients, the MA_{DAG} is not responsible for workflow execution but it only manages the scheduling phase.

3.2 Extended MA_{DAG} for static scheduling

While DIET was able to schedule workflows dynamically, it could not do static scheduling yet. We have thus extended the MA_{DAG} agent to make it able to apply a static schedule given by the user. From the user point of view, the client sends to the MA_{DAG} the usual files needed for the execution of the workflow, plus files describing the desired schedule:

- The Workflow Description File (WDF): the XML file describing the workflow as in any DIET execution of workflow,

- The Desired Schedule file (DSF): a file giving the desired schedule. Each line gives the machine attributed to a task, under the formulation <task> <server>. The order of the tasks gives their priority, the first ones having the highest priority. The static schedule is used to write the code of the target servers, hence this file has to be made available before launching execution,

- The Mapping File (MF): a file giving the equivalence between the name of the platform servers and their counterpart from the DSF. Each line gives a machine-server equivalence, under the formulation <machine> <server>.

DIET will then follow the given schedule.

On a more technical level, the agent responsible for the application of the given schedule is the MA_{DAG}. While it previously only determined in which order it should send the requests corresponding to each task to the master agent, and which tasks were ready to be sent, it now forces the master agent to send them to a specific server. In the traditional way of using DIET, it is not possible to specify, in the MA_{DAG}, which server will receive a given task: the MA_{DAG} leaves the duty of selecting a server to its master agent, only sending it the name of the services ready to be executed. To alleviate this limitation, we exploit the fact that the master agent uses the name of the service to find which server can run it. In order to let the user to be able to change the task-server attribution according to the information gathered by the DIET agents, we choose to have the servers declare their services twice, in two different ways. The first one is a generic declaration, as in any regular server for DIET, with a name identical to the one given in the XML file describing the workflow, and the same for all the servers able to run it. This allows the MA_{DAG} to get the list of all the servers available to run a given service corresponding to a task, and let some freedom so as to, for example, create a new functionality which will temper with the first schedule made offline. The second declaration is specific to the new functionality and has a name composed of the concatenation between the name of the service as described in the XML file and the name of the server. This local name is the one which will be used to send the request once the server will have been chosen. The order in which the tasks will be sent to the master agent is determined by the order in which they are given in the task-server attribution file (DSF).

3.3 Experiment-oriented tools

Given the large number of runs needed by the experiments, we have developed a couple of tools to automate the creation of the workflows meant to be executed on Grid'5000. We first used a home-made simulator [14] to generate the schedules with the selected algorithms. We then generated the corresponding DIET workflows needed for our experiments with a home-made workflow generator, OGMa [14].

The simulator is based on SimDag [22]. It generates both the static schedules given to DIET and the simulated results. Basically, the simulator needs a description of the platform, and of the workflow to be scheduled in a DAX format[5], and the parameters of the scheduling problem (budget, used algorithms, . . .). It creates a schedule, writes the mapping <task, VM > into a file intended for OGMa, in the order of their attribution, as determined by the chosen algorithm. Then it simulates with simDAG the execution of the given workflow on the given platform, using the calculated mapping. Finally, it calculates and writes in a file the resulting makespan, cost, and various other metrics.

Once the simulations done and the schedules calculated, we used OGMa to generate the elements needed for the experiments on Grid'5000. Concretely, OGMa uses the given DAX file describing the wanted workflow, along with information on the focused platform and simulations, and the static schedule, to write all files: the source files for the servers and the client, the configuration files needed for every DIET entities, placeholder files for the workflow, the DSF, the MF and the WDF.

We point out that, as DAX files do not give any details about the real content of the tasks or the files they describe, OGMa only creates placeholder files and tasks. If a user wants to use OGMa as a tool to generate the raw structure of the workflow, they will have to replace these placeholders with the actual code for the tasks.

We have used the new static scheduling functionality of DIET to compare the behavior on Grid'5000 of the scheduling algorithms described below in Section 4.

4 Budget-aware scheduling algorithms

In this section, we present the platform/application model that we have used in simulation, and the scheduling algorithms that we compare using the

[5] `https://pegasus.isi.edu/documentation/development/schemas.html`

DIET workflow engine. The material presented in this section is summarized from [5].

4.1 Workflow model

A workflow is represented with a DAG of stochastic tasks. Tasks are not preemptive and must be executed on a single processor[6]. Most workflow scheduling algorithms use as starting assumption that the exact number of instructions constituting a task is known in advance, so that its execution time is given accurately. However, this hypothesis is not always realistic. The number of instructions for a given task may strongly depend on the current input data, such as for image processing kernels. In our model, we only know an estimation of the number of instructions for each task. For lack of knowledge about the origin of time variations, we assume that all the parameters which determine the number of instructions forming a task are independent. This resulting number is the **weight** w_i of task T_i and follows a truncated Normal law with mean $\overline{w_i}$ and standard deviation $\sigma_i\overline{w_i}$:

$$w_i \sim \mathcal{N}(\overline{w_i}, \sigma_i\overline{w_i}) \tag{1}$$

Here σ_i is a parameter to control the standard deviation. To truncate, we draw randomly from the normal law until the result falls in the interval $[\overline{w_i} - \sigma_i\overline{w_i}, \overline{w_i} + \sigma_i\overline{w_i}]$. The value of $\overline{w_i}$ can be estimated (for example by sampling). Normal laws are ubiquitous in scientific applications [13] and therefore natural candidates to model the distribution of task weights. We truncate them to avoid negative values, as well as too large values. Note that the Pegasus generator also uses truncated normal laws [20].

Finally, to each dependency $(T_i, T_j) \in E$ is associated an amount of data of size $\mathit{size}(d_{T_i,T_j})$.

4.2 Platform model

Our model of Cloud platform mainly consists of a Cloud storage and processing units. To a great extent, it is based upon the offers of three big Cloud providers: Google Cloud[7], Amazon EC2[8] and OVH[9]. Given that Cloud providers propose a fault-tolerance service which ensures a very high

[6] This assumption is only for the sake of the presentation; it is easy to extend the approach to parallel tasks.

[7] `https://cloud.google.com/compute/pricing`

[8] `https://aws.amazon.com/ec2/pricing/on-demand/`

[9] `https://www.ovh.com/fr/public-cloud/instances/tarifs/`

availability of resources (in general over 99.97%[10]) as well as sufficient data redundancy, the Cloud storage and processing units are considered reliable and not subject to faults.

There is only one datacenter, used by all processing units. It is the common crossing point for all the data exchanges between processing units: these units do not interact directly. This model covers our needs in resilience and allows for the traceability of workflow execution. When a task T is to be executed on a VM v, all input data of T generated by one predecessor T' must be accessed from the Cloud storage, unless this data has been produced on the same VM v (meaning that T' had also been scheduled on v). The Cloud storage is also where the final generated data are stored before being transferred to the user. For simplicity, we consider that the Cloud storage bandwidth is large enough to feed all processing units, and to accommodate all submitted requests simultaneously, without any supplementary cost.

The processing units are VMs (Virtual Machines). They can be classified in different categories characterized by a set of parameters fixed by the provider. Some providers offer parameters of their own, such as the number of forwarding rules[11]. We only retain parameters common to the three providers Google, Amazon and OVH: a VM of category k has n_k processors, one processor being able to process one task at a time; a VM has also a speed s_k corresponding to the number of instructions that it can process per time unit, a cost per time-unit $c_{h,k}$ and an initial cost $c_{ini,k}$; all these VMs take an initial, and uncharged, amount of time t_{boot} to boot before being ready to process tasks. Already integrated in the schedule computing process, this starting time is thus not counted in the cost related to the use of the VM. Without loss of generality (even if the VM is paid for each used second), categories are sorted according to hourly costs, so that $c_{h,1} \leq c_{h,2} \cdots \leq c_{h,n_k}$. We expect speeds to follow the same order, but do not make such an assumption.

Altogether, the platform consists of a set of n VMs of k possible categories. Some simplifying assumptions make the model tractable while staying realistic: (i) we assume that the bandwidth is the same for every VM, in both directions, and does not change throughout execution; (ii) a VM is able to store enough data for all the tasks assigned to it: in other words, a VM will not have any memory/space overflow problem, so that every increase of the total makespan will be because of the stochastic aspect of the task weights; (iii) initialization duration is the same for every VM; (iv) data transfers take place independently of computations, hence do not have any impact on pro-

[10] `https://cloudharmony.com/status`

[11] `https://cloud.google.com/compute/pricing`

cessor speeds to execute tasks; (v) a VM executes at most one task at every time-step, but this task can be parallel and enroll many computing resources (hence the execution time of the task strongly depends upon the VM type).

We chose an "on-demand" provisioning system: it is possible to deploy a new VM during the workflow execution if needed. Hence VMs may have different startup times. A VM v is started at time $H_{start,v}$ and does not stop until all the data created by its last computed task have been transferred to the Cloud storage, at time $H_{end,v}$. VMs are allocated by continuous slots. If one wants discontinuous allocations, one may free the VM, then use a new one later, which at least requires sending all the data generated by the last processed task to the Cloud storage, and reloading all input data of the first task scheduled on that new VM before execution.

4.3 Scheduling costs and objective

Tasks are mapped to VMs and locally executed in the order given by the scheduling algorithm, such as those described in Section 4.4. Given a VM v, a task is launched as soon as (i) the VM is idle; (ii) all its predecessor tasks have been executed, and (iii) the output files of those predecessors mapped onto others VMs have been transferred to v via the Cloud storage.

Costs The cost model is meant to represent generic features out of the existing offers from Cloud providers (Google, Amazon, OVH). The total cost of the whole workflow execution is the sum of the costs due to the use of the VMs and of the cost due to the use of the Cloud storage $\mathcal{C}_{CS}$. The cost C_v of the use of a VM v of category k_v is calculated as follows:

$$C_v = (H_{end,v} - H_{start,v}) \times c_{h,k_v} + c_{ini,k_v} \tag{2}$$

There is a startup cost c_{ini,k_v} in Equation (2), and a term c_{h,k_v} proportional to usage duration $H_{end,v} - H_{start,v}$.

The cost for the Cloud storage is based on a cost per time-unit $c_{h,CS}$, to which we add a transfer cost. This transfer cost is computed with the amount of data transferred from the external world to the Cloud storage (size($d_{in,CS}$)), and from the Cloud storage to the outside world (size($d_{CS,out}$)). In other words, $d_{in,CS}$ corresponds to data that are input to entry tasks in the workflow, and $d_{CS,out}$ to data that are output from exit tasks. Letting $H_{start,first}$ be the moment when we book the first VM and $H_{end,last}$ be the moment when the data of the last processed task have entirely been sent to the Cloud storage, we define $H_{usage} = H_{end,last} - H_{start,first}$ as the total platform usage during the whole execution. We have:

$$\begin{aligned} C_{CS} = \quad & (\text{size}(d_{in,CS}) + \text{size}(d_{CS,out})) \times c_{tsf} \\ & + H_{usage} \times c_{h,CS} \end{aligned} \tag{3}$$

Altogether, the total cost is $C_{wf} = \sum_{v \in R_{VM}} C_v + C_{CS}$, where R_{VM} is the set of booked VMs during the execution.

Objective Given a budget $\mathcal{B}$, the objective is to minimize total execution time while respecting the budget:

$$\textsc{Minimize } H_{usage} \qquad \textsc{subject to } \mathcal{B} \geq C_{wf} \tag{4}$$

4.4 The HEFTBudg scheduling algorithm

In this section, we detail the design of HEFTBudg, a budget-aware extension of HEFT [24]. This extension accounts both for task stochasticity and budget constraints, while aiming at makespan minimization. Coping with task stochasticity is achieved by adding a certain quantity to the average task weight so that the risk of under-estimating its execution time is reasonably low, while retaining an accurate value for most executions. We use a conservative value for the weight of a task T, namely $\overline{w_T} + \sigma_T \overline{w_T}$.

Algorithm 1 Dividing the budget into tasks.

```
function divBudget($wf, \mathcal{B}_{calc}, \overline{s}, bw$)
    $W_{max} \leftarrow$ getMaxTotalWork($wf$)
    $d_{max} \leftarrow$ getMaxTotalTransfData ($wf$)
    for each $T$ of $wf$ do
        $budgPTsk[T] \leftarrow \mathcal{B}_{calc} \times \frac{\frac{\overline{w_T} + \sigma_T \overline{w_T}}{\overline{s}} + \frac{size(d_{pred,T})}{bw}}{\frac{W_{max}}{\overline{s}} + \frac{d_{max}}{bw}}$
    end for
    return $budgPTsk$
end function
```

Let us detail Algorithm 1: given the workflow wf, we first get the maximum of total work (getMaxTotalWork(wf)) and the total amount of data transfers (getMaxTotalTransfData (wf)) required to execute the workflow, and we reserve a fraction of the budget to cover the cost of the Cloud storage and VM initialization; then we divide what remains, $\mathcal{B}_{calc}$, into the workflow tasks. To estimate the fraction of budget to be reserved, assuming that $\mathcal{B}_{ini}$ denotes the initial budget:

- For the cost of the Cloud storage, we need to estimate the duration

$$H_{usage} = \ H_{end,last} \ - \ H_{start,first}$$

of the whole execution (see Equation (3)). To this purpose, we consider an execution on a single VM of the first (cheapest) category, compute the total duration $W_{max} = \sum_{T \in wf} (\overline{w_T} + \sigma_T)$ and let

$$H_{usage} = \frac{W_{max}}{s_1} + \frac{\text{size}(d_{in,CS}) + \text{size}(d_{CS,out})}{bw} \tag{5}$$

Altogether, we pay the cost of input/output data several times: with factor c_{tsf} for the outside world, with factor $c_{h,CS}$ for the usage of the Cloud storage (Equation (5)), and with factor $c_{h,1}$ during the transfer of data to and from the unique VM. However, there is no communication internal to the workflow, since we use a single VM.

- For the initialization of the VMs, we assume a different VM of the first category per task, hence we budget the amount $n \times c_{ini,1}$.

Combining these two choices is conservative: on the one hand, we consider a sequential execution, but account only for input and output data with the external world, eliminating all internal transfers during the execution; on the other hand, we reserve as many VMs as tasks, ready to pay the price for parallelism, at the risk of spending time and money due to data transfers during the execution. Altogether, we reserve the corresponding amount of budget and are left with $\mathcal{B}_{calc}$ for the tasks.

This reduced budget $\mathcal{B}_{calc}$ is shared among tasks in a proportional way: we estimate how much time $t_{calc,T}$ is required to execute each task T, transfer times included, and allocate the corresponding part of the budget in proportion to the whole for execution of the entire workflow $t_{calc,wf}$:

$$budgPTsk[T] = \frac{t_{calc,T}}{t_{calc,wf}} \times \mathcal{B}_{calc} \tag{6}$$

In Equation (6), we use $t_{calc,T} = \frac{\overline{w_T}+\sigma_T}{\overline{s}} + \frac{size(d_{pred,T})}{bw}$, where

$$size(d_{pred,T}) = \sum_{(T',T) \in E} size(d_{T',T}) \tag{7}$$

is the volume of input data of T from all its predecessors. Similarly, we use $t_{calc,wf} = \frac{W_{max}}{\overline{s}} + \frac{d_{max}}{bw}$, where $d_{max} = \sum_{(T_i,T_j) \in E} size(d_{T_i,T_j})$ is the total volume of data within the workflow. Computed weights ($\overline{w_T} + \sigma_T$ and

W_{max}) are divided by the mean speed $\overline{s}$ of VM categories, while data sizes ($size(d_{pred,T})$ and d_{max}) are divided by the bandwidth bw between VMs and the Cloud storage. Again, it is conservative to assume that all data will be transferred, because some of them will be stored in-place inside VMs, so there is here another source of over-estimation of the cost. On the contrary, using the average speed $\overline{s}$ in the estimation of the computing time may lead to an under-estimation of the cost when cheaper/slower VMs are selected.

Algorithm 2 Choosing the best host for each ready task.

```
function getBestHost($T, budgPTsk[T], \mathcal{P}, pot$)
    $\mathcal{B}_T \leftarrow budgPTsk[T] + pot$
    // initialisation: new host of cheapest category:
    $bestHost \leftarrow v$, where $v \in \text{New}_{\text{VM}}$ and $k_v = 1$
    $minEFT \leftarrow EFT_{T,bestHost}$
    for each $host$ of $(\text{Used}_{\text{VM}} \cup \text{New}_{\text{VM}})$ do
        if $((EFT_{T,host} < minEFT)$ and $(c_{T,host} <= \mathcal{B}_T))$ then
            $minEFT \leftarrow EFT_{T,host}$
            $bestHost \leftarrow host$
            $pot \leftarrow \mathcal{B}_T - c_{T,host}$
        end if
    end for
    return $bestHost, pot$
end function
```

This subdivided budget is then used to choose the best VM to host each ready task (see Algorithm 2): the best host for a task T on platform $\mathcal{P}$ will be the one providing the best EFT (Earliest Finish Time) for T, among those respecting the amount of budget $\mathcal{B}_T$ allocated to T. The platform $\mathcal{P}$ is defined as the set of host candidates, which consists of already used VMs plus one fresh VM of each category. For each host candidate $host$, either already used (set Used_{VM}) or new candidate (set New_{VM}), we first evaluate the time $t_{Exec,T,host}$ needed to have T executed (i.e., transfer of input data and computations) on $host$:

$$t_{Exec,T,host} = \delta_{new} \times t_{boot} + \frac{\overline{w_T} + \sigma_T \overline{w_T}}{s_{k_{host}}} + \frac{size(d_{in,T})}{bw} \tag{8}$$

In Equation (8), we introduce the boolean δ_{new} whose value is 1 if $host \in \text{New}_{\text{VM}}$ to account for its startup delay, and 0 otherwise. Also, some input data may already be present if $host \in \text{Used}_{\text{VM}}$, thus we use $size(d_{in,T})$ instead of $size(d_{pred,T})$ (see Equation (7)), defining $d_{in,T}$ as those input data not already present on $host$.

To compute $EFT_{T,host}$, the Earliest Finish Time of task T on host *host*, we account for its Earliest Begin Time $t_{begin,host}$ and add $t_{Exec,T,host}$. Then $t_{begin,host}$ is simply the maximum of the following quantities: (i) availability of *host*; (ii) end of transfer to the Cloud storage of any input data of T. The latter includes all data produced by a predecessor of T executed on another host; these data have to be sent to the Cloud storage before being re-emitted to *host*, since VMs do not communicate directly. There is a cost associated to these transfers, which we add to $t_{Exec,T,host} \times c_{h,host}$ to compute the total cost $c_{T,host}$ incurred to execute T on *host*. We do not write down the equation defining $t_{begin,host}$, as it is very similar to previous ones. Since we already subtracted from the initial budget everything except the cost of the use of the VMs themselves, getBestHost() can safely use $\mathcal{B}_T$ as the upper bound for the budget reserved for task T.

Algorithm 3 HEFTBudg.

```
function HEFTBudg($wf, \mathcal{B}_{calc}, \mathcal{P}$)
    $\overline{s} \leftarrow calcMeanSpeed(\mathcal{P})$
    $bw \leftarrow getBw(\mathcal{P})$
    $budgPTsk \leftarrow divBudget(wf, \mathcal{B}_{calc}, \overline{s}, bw)$
    ListT $\leftarrow getTasksSortedByRanks(wf, \overline{s}, bw, \overline{lat})$
    $pot, newPot \leftarrow 0$
    for each $T$ of ListT do
        $host \leftarrow getBestHost(T, budgPTsk[T], \mathcal{P}, newPot)$
        $pot \leftarrow newPot$
        $sched[T] \leftarrow host$
        $schedule(T, host)$
        $update(\text{Used}_{\text{VM}})$
    end for
    return ListT, $sched$
end function
```

The algorithm reclaims any unused fraction of the budget consumed when assigning former tasks: this is the role of the variable *pot*, which records any leftover budget in previous assignments. Finally, HEFTBudg (Algorithm 3) is the extension of the original HEFT algorithm. For some tasks, getBestHost() will not return the host with the smallest ETF, but instead the host with the smallest ETF among those that respect the allotted budget. The complexity of HEFTBudg is $O((n + e)p)$, where n is the number of tasks, e is the number of dependence edges, and p the number of enrolled VMs. This complexity is the same as for the baseline versions, except that p is not fixed *a priori*. In the worst case, $p = O(\max(n, k))$ because for each task we try all used VMs, whose count is possibly $O(n)$, and k new ones, one per category.

4.5 Other scheduling algorithms

Several budget-aware scheduling algorithms have been introduced in [5]:

- MinMinBudg, a budget-aware extension of MinMin [4, 11], which is the exact counterpart of HEFTBudg.
- HEFTBudg+ and HEFTBudg+Inv, which are refined (but more costly) variants of HEFTBudg that squeeze the most of any leftover budget to further decrease total execution time. The refined versions aim at exploiting the opportunity to re-schedule some tasks onto faster VMs, thereby spending any budget fraction leftover by the first allocation. These refined variants differ by the order in which tasks are considered, and recompute the schedule after processing each task. They are much more time-consuming than HEFTBudg, so there is a trade-off to find in terms of scalability.
- HEFTBudgMult, a trade-off version that reallocates the leftover budget in a single pass: it finds makespans slightly larger than those computed with HEFTBudg+, but with a time complexity close to HEFTBudg.

Two competitors to the new budget-aware algorithms described above have also been simulated in [5], for the sake of comparison. These are Budget Distribution with Trickling (BDT [1]) and Critical Greedy (CG [25]). Both BDT and CG schedule deterministic workflows, and CG does not take into account communication costs. In [25], CG also comes with a refined version CG+. BDT and CG/CG+ have been extended to fit the model, so as to enforce fair comparisons. Again, see [5] for a detailed description of all these algorithms.

5 Experimental framework

We use the new DIET functionality to execute the scheduling algorithms described in Section 4 on an actual platform, and we also run the corresponding experiments within our self-made simulator. We aim at an accurate comparison, hence we select all execution parameters very carefully so that both the experiments and the simulation obey the same model, or, at least, models that are as close as possible.

The real-life experiments are carried out using Grid'5000[12], a French national testbed supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. For each workflow execution, we used 31 homogeneous nodes from the `grisou` cluster of Nancy (Intel Xeon E5-2630 v3 CPU 2.4 GHz ×8 cores). The three different types of VMs (see Table 1) needed for the experiments have been emulated using a ratio on the number of operations made by a service. Hence, a VM which is two times slower than another one will execute twice as many computations.

VM parameters	
Categories Setup cost	$k = 3$ $c_{ini,=}\$0.00056$
Category 1 (Slow)	Speed $s_1 = 3.2$ Gflops Cost $c_{h,1} = \$0.118$ per hour
Category 2 (Medium)	Speed $s_2 = 6.4$ Gflops Cost $c_{h,2} = \$0.236$ per hour
Category 3 (Fast)	Speed $s_3 = 9.6$ Gflops Cost $c_{h,3} = \$0.354$ per hour
Cloud storage	
Cost per month	$c_{h,CS} = \$0.022$ per GB
Data transfer cost	$c_{tsf} = \$0.055$ per GB
Bandwidth	
bw	1GBps

Table 1: Parameters of the platform.

We used three types of workflows: MONTAGE, LIGO and CYBERSHAKE, generated with [20]. Given the large makespan of these workflows (e.g., 33 hours for one CYBERSHAKE of 60 tasks scheduled with CG/CG+ [5]), we only ran small instances with 30 tasks, but we point out that the new DIET functionality has no limitation in the size of the executed workflows.

5.1 Simulations

We used the simulator [14], described in Section 3.3, to obtain both the static schedules to be evaluated on Grid'5000 and the simulated results.

The characteristics of the platform used by the simulator (bandwidth: 1Gb/s, performances of the VMs used as slowest type: 3.2Gf) have been experimentally measured on the `grisou` cluster of Grid'5000.

[12] `https://www.grid5000.fr`

5.2 Grid'5000 experiments

As stated in Section 3.3, we generated the DIET workflows executed on Grid'5000 with OGMa, based on the very same workflows as the ones used during the simulation phase. Since we wanted to compare results obtained from simulations and real life experiments, we had to ensure that these results were comparable.

The first action item was to make sure that tasks have similar durations in the simulations and in the experiments. In the workflow description file, the amount of work of a task is given in seconds, when the amount of data transferred between two tasks is given in bytes. Thus for OGMa, a task consists of three phases: a phase during which the input files are read, a phase during which an amount of double floating-point additions is calculated based on the task duration given in the DAX file and the information provided about the used platform, and a phase during which the output files are written.

SimDAG uses a fixed (but arbitrary) number to calculate a number of flops based on the task duration in seconds given by the DAX file. This arbitrary number did not correspond to the VMs enrolled on Grid'5000. To preserve a similar ratio between the time used to move data $t_{data,T}$ and the time used to execute a task $t_{calc\_only,T}$ in the real setup and in the simulation, we modified the number of operations for each service according to the observed characteristics of the platform.

In the simulator, the amount of time $t_{calc\_only,T}$ used to execute the task T, composed of w_T operations, on a reference host of speed $s_{host,simu}$ operations per second, and without counting data transfers, is equal to

$$t_{calc\_only,T} = \frac{w_T}{s_{host,simu}}.$$

The amount of time $t_{data,T}$ to transfer all the data needed to execute T, for a total size of $size(d_{pred,T})$, with a bandwidth bw_{simu} is equal to

$$t_{data,T} = \frac{size(d_{pred,T})}{bw_{simu}}.$$

Hence the ratio to preserve is:

$$r_{simu} = \frac{w_T}{size(d_{pred,T})} \times \frac{bw_{simu}}{s_{host,simu}}.$$

Similarly, in the Grid'5000 execution, the ratio that we want to achieve for a bandwidth bw_{g5000} and a reference host speed $s_{host,g5000}$, with the same amount of transferred data $size(d_{pred,T})$ and a task T composed of $w_{T,g5000}$ operations ,is

$$r_{g5000} = \frac{w_{T,g5000}}{size(d_{pred,T})} \times \frac{bw_{g5000}}{s_{host,g5000}}.$$

Finally, we calculated the number of operations needed to execute the task T on a reference host of Grid'5000 as:

$$w_{T,g5000} = \frac{\frac{w_{T,simu}}{s_{host,simu}} \times bw_{simu}}{bw_{g5000}} \times s_{host,g5000}.$$

We used this number of operations to generate workflow tasks of appropriate duration.

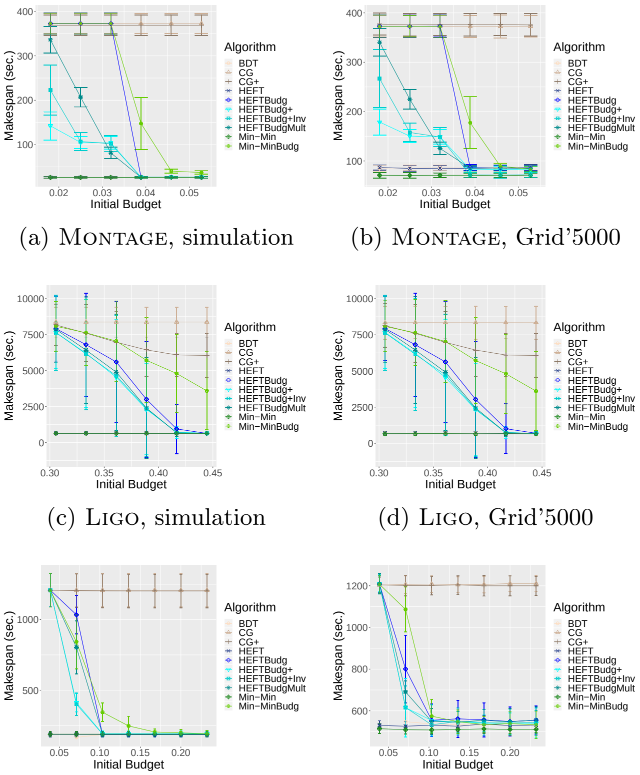
In the case of Ligo and Cybershake, instead of running an equally long workflow as described in the DAX files, we chose to generate one with the same shape (same tasks, dependencies between tasks, proportion between data transfers and amount of work for tasks), but whose malespan is made shorter, with the application of a fixed coefficient to decrease the time of execution of the workflows. As seen in Section 6, this has close to no impact on the performance of the algorithms.

As per the execution itself with DIET, we used one VM to run the client task, the master agent, the MA$_{DAG}$ agent and omniNames (the naming service on which DIET relies to operate communication between its components), and one VM per server.

5.3 Experimental campaign

We have generated schedules: (i) for three workflow types: Cybershake, Ligo and Montage; and (ii) for ten scheduling algorithms: MinMinBudg, HEFTBudg, HEFTBudgMult, HEFTBudg+, HEFTBudg+Inv, BDT, MinMin, CG, CG+ and HEFT. We have selected a range of budget values which have an actual impact on the makespan. For some of these budget values, a few algorithms are failing to produce valid schedules. Here we define a **valid** schedule as a schedule which successfully enforces the budget constraint.

We have executed, on Grid'5000 and on the simulator, 30 experiments per combination (budget × algorithm × workflow), and collected the makespan and cost of each execution. The costs are calculated as in [5], with prices adapted to the performances of the used VMs. The raw data, their treatment, and the whole experimentation setup, are available from [14].



(a) MONTAGE, simulation (b) MONTAGE, Grid'5000

(c) LIGO, simulation (d) LIGO, Grid'5000

(e) CYBERSHAKE, simulation (f) CYBERSHAKE, Grid'5000

Figure 2: Makespans for MONTAGE, CYBERSHAKE and LIGO workflows of 30 tasks, execution on simulation vs. Grid'5000.

6 Results

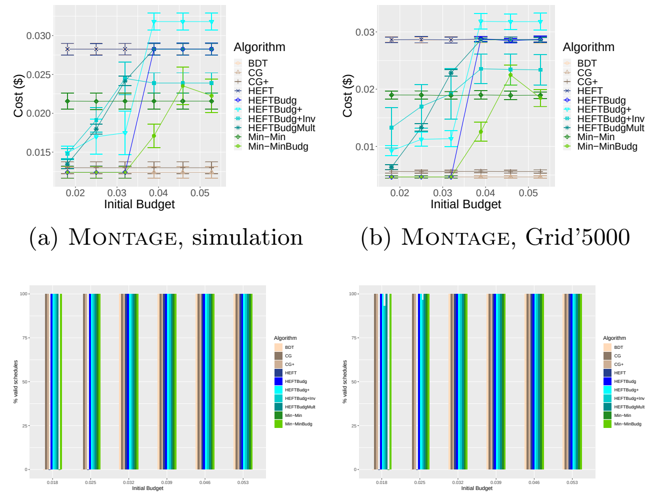
We first focus on the observation of makespan results from real life experiments compared to their simulation. Even results that spend more than the allocated budget are considered (in other words, these results are produced by non-valid schedules). Then we assess the accuracy of experimental costs conducted with the MONTAGE workflow, with a comparison between real life experiments and their simulation, and in this case we restrict to results under the given budget.

The experiments have been carried so that the generated workflows are equivalent to their simulated counterpart, with a fixed factor as only difference. For the sake of comparison, and to highlight how similar the obtained results are between the simulation and the execution on Grid'5000, we scale them in the figures: Montage makespans are 4.6 times lower than their simulation, Ligo ones are 49.06 times lower, and Cybershake ones 6.9 times lower. Original data is available in [14].

In Figure 2, we report the makespans obtained for each type of workflow as a function of the available budget. The first column represents the results obtained with the simulator. The second column represents the results obtained from the runs with DIET on Grid'5000. In most cases, the hierarchy of algorithms in the DIET executions is the same as the one in simulation. CG and CG+ obtain the highest makespans, and HEFT, BDT and MinMin the lowest ones. Among the budget-aware algorithms, MinMinBudg gets most of the time the highest makespans, but is inferior to the ones obtained with CG+. HEFTBudg obtains the second highest ones. HEFTBudg+ and HEFTBudg+Inv find the schedules with the lowest makespans. HEFTBudgMult schedules have makespans between HEFTBudg and HEFTBudg+ ones.

While the results of simulation and real execution are very similar for Montage and Ligo, there are some differences for Cybershake. We guessed that these differences might be due to file transfers, and we reran the simulations with an infinite bandwidth, but this had no impact. Still, overall, there is a pretty good correspondence between simulations and actual runs.

Next, we focus on Montage workflows, and in Figure 3, we report the costs and the percentage of valid solutions found for Montage executions. In Figures 3a and 3b, we see the cost of the execution of Montage workflows, both for simulations and real executions, and they match almost perfectly. With low budgets, two algorithms achieve very expensive schedules: BDT and HEFT. They are followed by MinMin. All the other budget-aware, algorithms, spend twice less budget to make a schedule. In addition, the higher the budget, the more optimization opportunities for budget-aware algorithms. For the highest budgets used for our experiments, we make the following observations: (i) HEFTBudg+ achieves the most expensive schedules, even higher than the ones from HEFT and BDT (but recall from Figure 2 that HEFTBudg+ needed a lower initial budget than HEFT and BDT to find valid makespans); (ii) HEFTBudg, HEFTBudgMult and HEFTBudg+Inv have a cost similar to HEFT (but achieve lower makespans); (iii) Similarly, MinMinBudg and MinMin schedules have similar costs (and lower makespans for MinMinBudg); and (iv) The cheapest schedules come from CG and CG/CG+ (but this is at the cost of a far larger makespan).



(a) Montage, simulation

(b) Montage, Grid'5000

(c) Montage, percentage of valid solutions, simulation

(d) Montage, percentage of valid solutions, Grid'5000

Figure 3: Costs for Montage workflows of 30 tasks, execution on Grid'5000 vs. simulation, and percentage of valid solutions.

Figures 3c and 3d show the percentage of valid schedules found by each algorithm, from simulations (left) and real-life experiments (right). We observe that the only difference is for HEFTBudg+Inv on the two lowest budgets, which does not give a 100% valid schedules in real-life. We know from [5] that this algorithm refines its schedule, leaving only a small leftover budget, which explains the difference. On each graph, we see that for the lowest budget, BDT, HEFT and MinMin give no valid schedule, and for the second lowest budget, BDT and HEFT still do not. All the other algorithms give 100% valid solutions.

7 Conclusion

In this paper, we have introduced a new scheduling functionality for DIET, and provided the user with a set of tools to implement, and experiment with, static scheduling algorithms for workflows. We then used this new functionality to compare the executions of ten static algorithms for scientific applications from the Pegasus benchmark [10], using both a simulator designed with simDAG and the testbed platform Grid'5000. Both types of experiments gave similar results, validating our new functionality.

Further work will be devoted to better understand the behavior of budget-aware algorithms on larger and more diverse workflows, using the insights gained from both the similarities and differences found in simulations and actual executions.

References

[1] V. Arabnejad, K. Bubendorfer, and B. Ng. Budget distribution strategies for scientific workflow scheduling in commercial clouds. In IEEE 12th Int. Conf. on e-Science (e-Science), pages 137–146, Oct 2016.

[2] M. Beauchemin. Apache airflow project, 2014.

[3] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In SC'08 Workshop: The 3rd Workshop on Workflows in Support of Large-scale Science (WORKS08) web site, Austin, TX, Nov. 2008. ACM/IEEE.

[4] T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. P. Robertson, M. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. Journal of Parallel and Distributed Computing, 61(6):810–837, 2001.

[5] Y. Caniou, E. Caron, A. K. W. Chang, and Y. Robert. Budget-aware scheduling algorithms for scientific workflows on IaaS cloud platform. Research Report 9088, INRIA, Aug. 2017. To appear in CCPE, 2021.

[6] E. Caron. Contribution to the management of large scale platforms: the DIET experience. HDR (Habilitation 'a Diriger les Recherches), École Normale Supérieure de Lyon, Oct.6 2010. hal number tel-00629060.

[7] E. Caron and F. Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. International Journal of High Performance Computing Applications, 20(3):335–352, 2006.

[8] E. Caron, F. Desprez, T. Glatard, M. Ketan, J. Montagnat, and D. Reimert. Workflow-based comparison of two distributed computing infrastructures. In Workflows in Support of Large-Scale Science (WORKS10), New Orleans, November 14 2010. In Conjunction with Supercomputing 10 (SC'10), IEEE. hal-00677820.

[9] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow Management in Condor. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, Workflows for e-Science, pages 357–375. Springer, 2007.

[10] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus: a workflow management system for science automation. Future Generation Computer Systems, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[11] P. Ezzatti, M. Pedemonte, and A. Martín. An efficient implementation of the min-min heuristic. Comput. Oper. Res., 40(11), 2013.

[12] A. Ilyushkin, A. Bauer, A. V. Papadopoulos, E. Deelman, and A. Iosup. Performance-feedback autoscaling with budget constraints for cloud-based workloads of workflows. CoRR, abs/1905.10270, 2019.

[13] K. Kim and G. Shevlyakov. Why gaussianity? IEEE Signal Processing Magazine, 25(2):102–113, 2008.

[14] A. Kong Win Chang. `https://graal.ens-lyon.fr/~achang/ccgrid2021/`.

[15] M. Krämer. Capability-based scheduling of scientific workflows in the cloud. In S. Hammoudi, C. Quix, and J. Bernardino, editors, DATA, pages 43–54. SciTePress, 2020.

[16] C. Lin and S. Lu. Scheduling scientific workflows elastically for cloud computing. In Cloud Computing (CLOUD), 2011 IEEE International Conference on, pages 746–747. IEEE, 2011.

[17] J. Liu, J. Ren, W. Dai, D. Zhang, P. Zhou, Y. Zhang, G. Min, and N. Najjari. Online multi-workflow scheduling under uncertain task execution time in iaas clouds. IEEE Transactions on Cloud Computing, pages 1–1, 2019.

[18] S. Makhlouf and B. Yagoubi. Data-aware scheduling strategy for scientific workflow applications in iaas cloud computing. International Journal of Interactive Multimedia and Artificial Intelligence, InPress:1, 01 2018.

[19] R. Mitchell, L. Pottier, S. Jacobs, R. F. d. Silva, M. Rynge, K. Vahi, and E. Deelman. Exploration of workflow management systems emerging features from users perspectives. In 2019 IEEE International Conference on Big Data (Big Data), pages 4537–4544, 2019.

[20] Pegasus. Code for the Pegasus generator. `https://github.com/pegasus-isi/WorkflowGenerator`, 2020.

[21] T. Risset and Y. Robert. Synthesis of processor arrays for the algebraic path problem. Parallel Processing Letters, 1(1):19–28, Sept. 1991.

[22] SimDag. Programming environment for DAG applications. `http://simgrid.gforge.inria.fr/simgrid/3.13/doc/group__SD__API.html`, 2017.

[23] S. Smanchat and K. Viriyapant. Taxonomies of workflow scheduling problem and techniques in the cloud. Future Generation Computer Systems, 52:1–12, 2015.

[24] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distributed Systems, 13(3):260–274, 2002.

[25] C. Q. Wu, X. Lin, D. Yu, W. Xu, and L. Li. End-to-end delay minimization for scientific workflows in clouds under budget constraint. IEEE Transactions on Cloud Computing, 3(2):169–181, April 2015.

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

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399