# Budget Constrained Resource Allocation for Non-Deterministic Workflows on a IaaS Cloud

Eddy Caron, Frédéric Desprez, Adrian Muresan, Frédéric Suter

# Budget Constrained Resource Allocation for Non-Deterministic Workflows on a IaaS Cloud

Eddy Caron[1], Frédéric Desprez[1], Adrian Muresan[1], Frédéric Suter[2]

[1]UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668
46, allée d'Italie, 69364 Lyon Cedex 7, France

[2]IN2P3 Computing Center, CNRS, IN2P3
43, bd du 11 novembre 1918, 69622 Villeurbanne, France

RESEARCH REPORT
N° 7962
May 2012
Project-Team GRAAL

ISSN 0249-6399 ISRN INRIA/RR--7962--FR+ENG

# Budget Constrained Resource Allocation for Non-Deterministic Workflows on a IaaS Cloud

Eddy Caron*[1], Frédéric Desprez†[1], Adrian Muresan‡[1], Frédéric Suter§[2]

[1]UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668
46, allée d'Italie, 69364 Lyon Cedex 7, France

[2]IN2P3 Computing Center, CNRS, IN2P3
43, bd du 11 novembre 1918, 69622 Villeurbanne, France

Project-Team GRAAL

Research Report n° 7962 — May 2012 — 27 pages

**Abstract:** Many scientific applications are described through workflow structures. Due to the increasing level of parallelism offered by modern computing infrastructures, workflow applications now have to be composed not only of sequential programs, but also of parallel ones. Cloud platforms bring on-demand resource provisioning and pay-as-you-go payment charging. Then the execution of a workflow corresponds to a certain budget. The current work addresses the problem of resource allocation for non-deterministic workflows under budget constraints. We present a way of transforming the initial problem into sub-problems that have been studied before. We propose two new allocation algorithms that are capable of determining resource allocations under budget constraints and we present ways of using them to address the problem at hand.

**Key-words:** resource allocation, scheduling, PTG, workflow

* Eddy.Caron@ens-lyon.fr
† Frederic.Desprez@ens-lyon.fr
‡ Adrian.Muresan@ens-lyon.fr
§ fsuter@cc.in2p3.fr

# Allocation sous contraintes de budget de workflows non-déterministes pour Cloud IaaS

**Résumé :** De nombreuses applications scientifiques sont décrites sous la forme de workflows. Du fait de l'accroissement du niveau de parallélisme offert par les infrastructures de calcul modernes, de telles applications doivent désormais être composées non seulement de programmes séquentiels mais aussi de programmes parallèles. Les Clouds offrent le provisionnement de ressources à la demande ainsi qu'une facturation à l'utilisation. L'exécution d'un workflow correspond alors à un certain budget. Dans cet article, nous considérons le problème de l'allocation de ressources à un workflow non déterministe en présence de contraintes de budget. Nous présentons une façon de transformer le problème initial en une série de sous-problèmes qui ont été largement étudiés. Nous proposons deux algorithmes originaux qui peuvent déterminer des allocations de ressources sous contrainte de budget. Nous détaillons également comment les utiliser pour résoudre le problème initial.

**Mots-clés :** Allocation de ressources, ordonnancement, PTG, workflow

## Contents

**1 Introduction** **3**

**2 Related Work** **4**

**3 Problem Statement** **5**
3.1 Platform and Application Models . . . . . . . . . . . . . . . . . . . 5
3.2 Metrics and Problem Statement . . . . . . . . . . . . . . . . . . . . 7

**4 Allocating a Non-Deterministic Workflow** **8**
4.1 Splitting the Workflow . . . . . . . . . . . . . . . . . . . . . . . . . 8
4.2 Distributing Budget to Sub-Workflows . . . . . . . . . . . . . . . . 9
4.3 Determining PTG allocations . . . . . . . . . . . . . . . . . . . . . 11
4.4 Scheduling and workflow execution . . . . . . . . . . . . . . . . . . 18

**5 Experimental evaluation** **19**
5.1 Experimental methodology . . . . . . . . . . . . . . . . . . . . . . 19
5.2 Platform description . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.3 Comparison of running times . . . . . . . . . . . . . . . . . . . . . 20
5.4 Simulation results . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**6 Conclusion and Future Work** **25**

## 1 Introduction

Many scientific applications from various disciplines are structured as *workflows*. Informally, a workflow can be seen as the composition of a set of basic operations that have to be performed on a given input set of data to produce the expected scientific result. The interest for workflows mainly comes from the need to build upon legacy codes that would be too costly to rewrite. Combining existing programs is also a way to lead to new results that would not have been found using each component alone. For years, such program composition was mainly done by hand by scientists, that had to run each program one after the other, manage the intermediate data, and deal with potentially tricky transitions between programs. The emergence of Grid Computing and the development of complex middleware components [6, 7, 9, 11, 12, 15, 17] automated this process.

The evolution of architectures with more parallelism available, the generalization of GPU, and the main memory becoming the new performance bottleneck, motivate a shift in the way scientific workflows are programmed and executed. A way to cope with these issues is to consider workflows composing not only sequential programs but also parallel ones. This allows for the simultaneous exploitation of both the task- and data-parallelisms exhibited by an application. It is thus a promising way toward the full exploitation of modern architectures. Each step of a workflow is then said to be *moldable* as the number of resources allocated to an operation is determined at scheduling time. Such workflows are also called Parallel Task Graphs (PTGs).

In practice, some applications cannot be modeled by classical workflow or PTG descriptions. Fur such applications the models are augmented with special semantics that allow for exclusive diverging control flows or repetitive flows.

This leads to a new structure called a *non-deterministic* workflow. For instance, we can consider the problem of gene identification by promoter analysis [2, 19] as described in [12], or the GENIE (Grid ENabled Integrated Earth) project that aims at simulating the long term evolution of the Earth's climate [14].

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/her application on a virtually infinite supply of resources. While the elasticity provided 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, that imposes certain constraints on the scheduling process. In this work we detail a first step to address this scheduling problem in the case of non-deterministic workflows. Our main contribution is the design of an original allocation strategy for non-deterministic workflows under budget constraints. We target a typical IaaS Cloud and adapt some existing scheduling strategies to the specifics of such an environment in terms of resource allocation and pricing.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes our application and platform models, and gives a precise problem statement. Section 4 details the proposed algorithm to allocate resources to non-deterministic workflows on an IaaS Cloud. Finally, Section 6 summarizes our contribution and presents some future work.

## 2 Related Work

The problem of scheduling workflows has been widely studied by the aforementioned workflow management systems. Traditional workflows consists in a deterministic DAG structure whose nodes represent compute tasks and edges represent precedence and flow constraints between tasks. Some workflow managers support conditional branches and loops [5], but neither of them target elastic platforms such as IaaS Clouds nor address their implications.

Several algorithms have been proposed to schedule PTGs, *i.e.,* deterministic workflows made of moldable tasks, on various non-elastic platforms. Most of them decompose the scheduling in two phases: (i) determine a resource allocation for each task; and (ii) map the allocated tasks on the compute resources. Among the existing algorithms, we based the current work on the CPA [16] and biCPA [8] algorithms. We refer the reader to [8] for details and references on other scheduling algorithms.

The flexibility provided by elastic resource allocations offers great improvement opportunities as shown by the increasing body of work on resource management for elastic platforms. In [10], the authors give a proof of concept for a chemistry-inspired scientific workflow management system. The chemical programming paradigm is a nature-inspired approach for autonomous service coordination [18]. Theirs results make this approach encouraging, but still less performing than traditional workflow management systems. In contrast to the current work, they do not aim at conditional workflows or budget constraints. An approach to schedule workflows on elastic platforms under budget constraints is given in [13], but is limited to workflows without any conditional structure.

## 3 Problem Statement

### 3.1 Platform and Application Models

An IaaS Cloud can be seen as a virtually infinite set of resources that are reserved and instantiated by users according to their needs. We consider that users have access to a *catalog* that comprises different types of resources, each corresponding to a unique combination of characteristics. Such a catalog is inspired by the offers of major providers such as Amazon EC2 [3]. A resource, or virtual machine instance, $vm$, can be described by:

- A number of equivalent virtual CPUs, $nCPU$. The number of virtual CPUs does not correspond to the number of physical CPUs in the instance, but allows users to easily compare the relative performance of different instances;
- A computing speed per virtual CPU, $s$. This corresponds to the amount of computing operations a single CPU can process per second.
- A monetary cost per running hour, $cost$, expressed in a currency-independent manner. As most providers do, we also consider that each started hour has to be entirely paid even when not fully used. This cost is then proportional to the number of full hours the instance runs since it becomes usable.

In our study, we consider that every virtual CPU in the IaaS Cloud have the same computing speed. Instances of the same type are then homogeneous, while the complete catalog is a heterogeneous set of resources. Thus, we do not include this speed in our formal definition of the catalog $\mathcal{C}$ that is

$$\mathcal{C} = \{vm_i = (nCPU_i, cost_i) | i \geq 1\}.$$

We also consider that a virtual CPU can communicate with several other virtual CPUs simultaneously under the *bounded multi-port* model. All the concurrent communication flows share the bandwidth of the communication link that connects this CPU to the remaining of the IaaS Cloud.

Our workflow model is inspired by previous work [14, 1]. We define a non-deterministic workflow as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i | i = 1, \ldots, V\}$ is a set of $V$ vertices and $\mathcal{E} = \{e_{i,j} | (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of $E$ edges representing precedence and flow constraints between tasks. Without loss of generality we assume that $\mathcal{G}$ has a single entry task and a single exit task. The vertices in $\mathcal{V}$ can be of different types. A **Task** node represents a (potentially parallel) computation. Such nodes can have any number of predecessors, *i.e.,* tasks that have to complete before the execution of this task can start, and any number of successors, *i.e.,* tasks that wait for the completion of this task to proceed. Traditional deterministic workflows are made of task nodes only. The relations between a task node and its predecessors and successors can be represented by control structures, that we respectively denote by **AND-join** and **AND-split** transitions.

Task nodes are moldable and can be executed on any numbers of virtual resource instances. We denote by $Alloc(v)$ the set of instances allocated to task $v$ for its execution. The total number of virtual CPUs in this set is then:

$p(v) = \sum_j nCPU_j | vm_j \in Alloc(v)$. It allows us to estimate $T(v, Alloc(v))$ the execution time of task $v$ if it were to be executed on a given allocation. In practice, this time can be measured via benchmarking for several allocations, or it can be calculated via a performance model. In this work, we rely on Amdahl's law. This model claims that the speedup of a parallel application is limited by its strictly serial part $\alpha$. The execution time of a task is given by

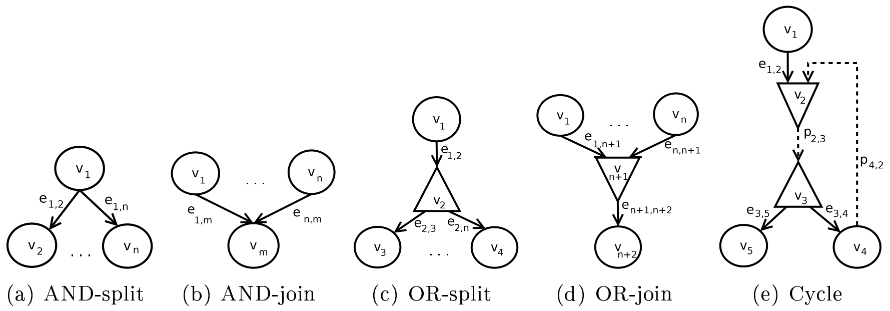
$$T(v, Alloc(v)) = \left(\alpha + \frac{(1-\alpha)}{p(v)}\right) \times T(v, 1),$$

where $T(v, 1)$ is the time needed to execute task $v$ on a single virtual CPU. The overall execution time of $\mathcal{G}$, or *makespan*, is defined as the time between the beginning of $\mathcal{G}$'s entry task and the completion of $\mathcal{G}$'s exit task. The total number of CPUs needed to achieve this makespan is $p = \sum_{i=1}^{V} p(v_i)$.

In our model, we consider that each edge $e_{i,j} \in \mathcal{E}$ has a weight, which is the amount of data, in bytes, that task $v_i$ must send to task $v_j$. We do not impose any type of restrictions for inter-task communications. The actual communication time may be higher than the time needed to transfer the data, as the source and destination tasks might be mapped to a different number of virtual resources, which might cause an overhead.

To model the non-deterministic behavior of the considered workflows, we add the following control nodes to our model. A **OR-split** node has a single predecessor and any number of successors, that represent mutually-exclusive branches of the workflow. When the workflow execution reaches an OR-split node, it continues through only one of the successors. The decision of which successor to run is taken at runtime. Then in the scheduling phase, all the sub-workflows deriving from an OR-split node have to be considered as equally potential execution paths. Conversely an **OR-join** node has any number of predecessors and a single successor. If any of the parent sub-workflows reaches this node, the execution continues with the successor.

Finally, our model of non-deterministic workflows can also include **Cycle** constructs. This is an edge joining an OR-split node and one OR-join ancestor. A cycle must contain at least one OR-join node to prevent deadlocks. Figure 1 gives a graphical representation of these control nodes and constructs.



(a) AND-split (b) AND-join (c) OR-split (d) OR-join (e) Cycle

Figure 1: Non-deterministic workflow control nodes and constructs.

Figure 1(e) is a simple representation of the Cycle construct. $p_{2,3}$ and $p_{4,2}$ are not edges of the workflow, but paths leading from $v_2$ to $v_3$ and from $v_4$ to $v_2$ respectifely. These paths are a weak constraint that ensure the creation of a cycle in the graph, in combination with the OR-join and OR-split nodes $v_2$ and $v_4$. However, a Cycle can contain any number of OR-split or OR-join nodes and even an unbound number of edges leading to other parts of the workflow.

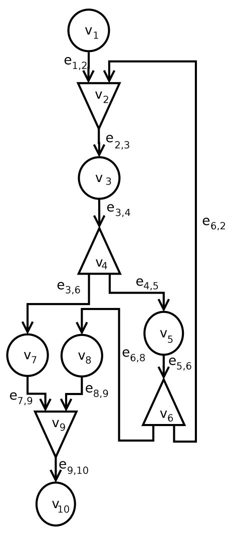


Figure 2: A more complex workflow example.

We give a more complex example of functional workflow in Figure 2, in which the path deriving from the edge $e_{6,2}$ comprises a OR-split node ($v_4$). This implies that the Cycle construct does not determine the number of iterations of the cycle path by itself, as in a loop construct for instance. Decisions taken at runtime for $v_4$ may make the execution flow exit the cycle before reaching $v_6$.

## 3.2 Metrics and Problem Statement

We consider the problem of determining allocations for a single non-deterministic workflow on an IaaS Cloud. It amounts to allocate resource instances to the tasks of this workflow so as to minimize its makespan while respecting a given budget constraint. Targeting an IaaS Cloud indeed implies such a constraint, as using more resources is likely to lead to smaller makespans but also increases the monetary cost associated to the execution of the workflow. An additional issue is to deal with the non-determinism of the considered workflows. At scheduling time, all the possible execution paths have to be considered. But at runtime, some sub-workflows will not be executed, due to the OR-split construct, while others may be executed several times, due to the Cycle construct. This raises some concerns relative to the respect of the budget constraint. Our approach is to decompose the workflow into a set of deterministic sub-workflows with non-deterministic transitions between them. Then, we fall back to the well studied problem of determining allocations for multiple Parallel Task Graphs (PTGs).

In the following we define the makespan as $C = \max_i C(v_i)$ where $C(v_i)$ is the finish time of task $v_i$. We denote by $B$ the budget allocated to the execution of the original workflow and by $B^i$ the budget allocated to the $i^{th}$ sub-workflow. These budgets are expressed in a currency-independent manner.

Finally, $Cost^i$ is the cost of a schedule $\mathcal{S}^i$ built for the $i^{th}$ sub-workflow on a dedicated IaaS Cloud. It is defined as the sum of the costs of all the resource instances used during the schedule. Due to the pricing model, we consider all started hour as fully paid.

$$Cost^i = \sum_{\forall vm_j \in \mathcal{S}^i} \lceil T_{end_j} - T_{start_j} \rceil \times cost_j,$$

where $T_{start_j}$ is the time when $vm_j$ is launched and $T_{end_{i=j}}$ the time when this resource instance is stopped.

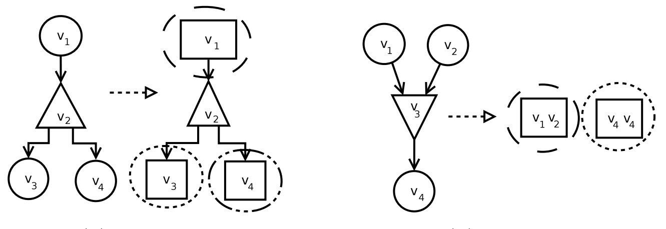
# 4 Allocating a Non-Deterministic Workflow

Our algorithm is decomposed in three steps: (i) Split the non-deterministic workflow into a set of deterministic PTGs; (ii) Divide the budget among the resulting PTGs and (iii) Determine allocations for each PTG.The following sections details these steps. We also discuss some runtime issues.

## 4.1 Splitting the Workflow

Transforming a non-deterministic workflow into a set of PTGs amounts to extract all the sequences of task nodes without any non-deterministic construct. A similar approach to decompose a workflow into smaller parts is taken by DagMan [6]. It allows users to split nested workflows by hand and is considered as part of the workflow definition.

Figure 3 shows how we extract sub-workflows in presence of OR-split and OR-join nodes. For the sake of simplicity we have omitted edge labels in this figure. These control nodes define boundaries between sub-workflows and do not belong to any of them. An OR-split node leads to $n + 1$ sub-workflows, one ending with the predecessor of the node and $n$ starting with each of the successors of the OR-split node. If two OR-split nodes share a common successor, we consider the two resulting sub-workflows as different, even though they have the same structure. Indeed these sub-workflows come from different non-deterministic transitions and therefore different contexts.



(a) OR-split (b) OR-join

Figure 3: Extracting sub-workflows from OR-split and OR-join nodes.

Splitting a workflow that contains an OR-join node can lead to as many sub-workflows as there were predecessor sub-workflows of the OR-join node. The successors of the OR-join node are replicated for all of its predecessors, including the ones that are part of the same sub-workflow. It is worth noting that OR-join nodes do not actually lead to the creation of new sub-workflows since they do not have a non-deterministic nature and therefore they do not lead to non-deterministic transitions. What they actually do is preserve the number of sub-workflows that they have from their inwards transitions.

Extracting sub-workflows from a Cycle node is more complex as shown in Figure 4. Here we extract three sub-workflows. Two of them include an instance of task $v_3$. One comes as a result of the execution of task $v_1$, while the other derives from following the cycle branch. Task $v_5$ is then the predecessor of this second instance.

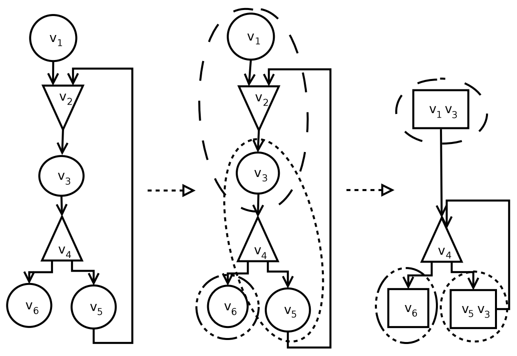


Figure 4: Extracting sub-workflows with regard to a Cycle construct.

Figure 5 details how we decompose the complex workflow given in Figure 2. It is worth noting that a Cycle constructs does not necessarily correspond to a unique sub-workflow. In this example, the Cycle $e_{6,2}$ is split into two different sub-workflows $v_3$ and $v_5$ that both belong to the cycle path. This will have an impact on budget distribution as detailed in the next section.

## 4.2 Distributing Budget to Sub-Workflows

As we target an IaaS Cloud, we have to decide how much money we can dedicate to each sub-workflow obtained after the split of the original application to determine its resource allocation. Because of the non-deterministic transitions between sub-workflows, we first have to estimate the odds to execute each of them. Moreover, as cycle paths may comprise several sub-workflows, we have to estimate how many times each sub-workflow could be executed at runtime.

Each sub-workflow, apart from the entry sub-workflow, has one and only one non-deterministic transition that triggers its execution. This is the transition from its parent OR-split node to its starting task. We can therefore conclude that the number of executions of a sub-workflow is described completely by the number of transitions of the edge connecting its parent OR-split to its start node. We model this behavior by considering that the number of transitions of each outwards edge of an OR-split, and therefore the number of executions of a sub-workflow $\mathcal{G}^i$ is described by a random variable according to a distinct *normal distribution* $D^i$. Moreover we use a parameter that express the *Confidence* the algorithm has that a given sub-workflow will not be executed more than a

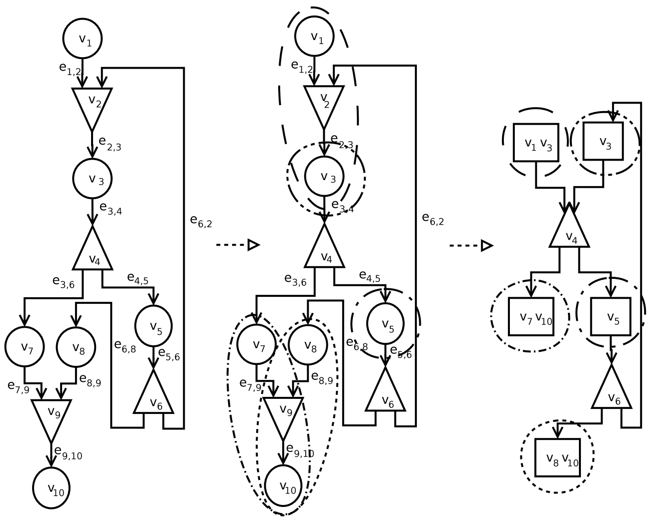


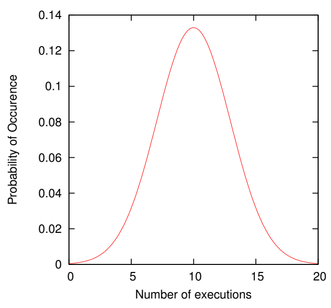
Figure 5: Extracting sub-workflows from a more complex workflow.

certain number of time. This parameter takes its value in the $[0, 1)$ interval. This way, we aim at guaranteeing that the whole workflow will be able to finish while respecting the budget constraint. More formally, the expected maximum number of executions of a $\mathcal{G}^i$ is

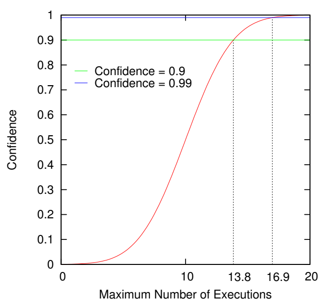
$$nExec^i \leftarrow CDF^{-1}(D^i)(Confidence)$$

where $CDF^{-1}(D^i)$ is the reverse *Cumulative Distribution Function* (CDF) for distribution $D^i$. Figures 6(a) and 6(b) illustrate our approach.

Figure 6(a) displays the normal distribution $\mathcal{N}(10, 3)$ of a random variable. The distribution median is $\mu = 10$ and its variance is $\sigma^2 = 3$. In our context, it correspond to the probability that the sub-workflow execution modeled by this random variable is repeated a certain number of times.



(a) Normal distribution $\mathcal{N}(10, 3)$.



(b) CDF of $\mathcal{N}(10, 3)$.

Figure 6: Estimation of the maximum number of executions of a sub-workflow, described by a normal distribution, with a certain confidence.

Figure 6(b) shows the CDF of this distribution. It allows to estimate, for a given confidence, how many time we will repeat the considered sub-workflow *at* *most.* For instance, with a confidence of 0.9 (or 90%), this sub-workflow is likely to not be executed more than 13.8 times. With a higher confidence of 0.99 (or 99%), this estimation raises up to 16.9 executions at most.

This estimation of the number of times a sub-workflow could be executed is not the only metric to consider to distribute the budget as best as possible. Indeed, it may be more important to give an important share of the budget to a sub-workflow with many time-consuming tasks that may be executed only once than to a sub-workflow with a few short tasks that is repeated several times. To find a good balance, we include the contribution of a sub-workflow with regard to the whole application in the determination of the budget distribution. We determine the contribution $\omega^i$ of sub-workflow $\mathcal{G}^i$ as the sum of the average execution times of its tasks multiplied by the number of times this sub-workflow could be executed. As the target platform is virtually infinite, we compute the average execution time of a task over the set of resource instances in the catalog $\mathcal{C}$. This allows us to take the speedup model into account, while reasoning on a finite set of possible resource allocations. We denote by $\omega^*$ the sum of the contribution made by all the sub-workflows.

**Algorithm 1** Share_Budget($B, \mathcal{G}, Confidence$)

```
1: $\omega^* \leftarrow 0$
2: for all $\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i) \subseteq \mathcal{G}$ do
3:     $nExec^i \leftarrow CDF^{-1}(D^i, Confidence)$
4:     $\omega^i \leftarrow \sum_{v_j \in \mathcal{V}^i} \left( \frac{1}{|\mathcal{C}|} \sum_{vm_k \in \mathcal{C}} T(v_j, vm_k) \right) \times nExec^i$
5:     $\omega^* \leftarrow \omega^* + \omega^i$
6: end for
7: for all $\mathcal{G}^i \subseteq G$ do
8:     $B^i \leftarrow B \times \frac{\omega^i}{\omega^*} \times \frac{1}{nExec^i}$
9: end for
```

Algorithm 1 describes how we distribute the global budget $B$ among the sub-workflows. Once we have estimated the number of execution of each workflow and its relative contribution, the budget $B^i$ assigned to one iteration of the sub-workflow $\mathcal{G}^i$ is simply obtained by multiplying the global budget by the ratio $\omega^i/\omega^*$ and dividing by the estimated number of executions of the workflow $nExec^i$ (line 8).

## 4.3 Determining PTG allocations

Once the non-deterministic workflow has been split into a set of deterministic sub-workflows, and that a budget has been assigned to each sub-workflow, our algorithm has to find an allocation for each of them. In other words, we have to determine which combination of virtual instances from the resource catalog leads to the best compromise between the reduction of the makespan and the monetary cost for each sub-workflow, *i.e.,* a PTG. We base our work upon the allocation procedures of seminal two-step algorithms, named CPA [16] and biCPA [8], that were designed to schedule PTGs on homogeneous commodity clusters. We adapt these procedures to the specifics of IaaS Cloud platforms.

As the biCPA algorithm is an improvement of the original CPA algorithm, we start by briefly explaining the common principle of their respective allocation procedures. It starts by allocating one CPU to each task in the PTG. Then it iterates to allocate one extra CPU to the task that belongs to the critical path of the application and benefits the most of it. The procedure stops when the average work $T_A$ becomes greater than the length of the critical path $T_{CP}$. The definition of the average work used by the CPA algorithm was

$$T_A = \frac{1}{P}\sum_{i=1}^{|\mathcal{V}^i|} W(v_i),$$

where $W(v_i)$ is the work associated to task $v_i$, *i.e.,* the product of its execution time by the number of CPUs in its allocation, and $P$ the total number of CPUs in the target compute cluster. In biCPA, the value of $P$ is iterated over from 1 to the size of the target compute cluster and its semantics is changed to represent the total number of CPUs that any task can have allocated to it.

The definition of the length of the critical path was

$$T_{CP} = max_i BL(v_i)$$

where $BL(v_i)$ represents the *bottom level* of task $v_i$ *i.e.,* its distance until the end of the application. For the current work we keep this definition for $T_{CP}$.

On an IaaS Cloud, the size of the target platform is virtually infinite. Then it is impossible to use such a definition that includes a total number of CPUs. Instead, we propose to reason in terms of budget and average cost of an allocation. Moreover, the pricing model implies that each started hour is paid, even though the application has finished its execution. Then, some *spare time* may remain on a virtual resource instance at the end of an execution.

When building an allocation, we don't know yet in which order the tasks will be executed. Then we cannot make any strong assumption about reusing spare time left behind after executing a task. As we aim at building an allocation for $\mathcal{G}^i$ that costs less than $B^i$, a conservative option would be to consider that this spare time is never used. This corresponds to always overestimating the cost of the execution of a task by rounding its execution time up to the end of the last started hour. Then we define this cost as

$$cost(v_i) = \lceil T(v_i, Alloc(v_i))\rceil \times \sum_{vm_j \in Alloc(v_i)} cost_j.$$

This, in turn, leads us to a first adapted version of the definition of $T_A$

$$T_A^{over} = \frac{1}{B'} \times \sum_{j=1}^{|\mathcal{V}^i|} \left(T(v_j, Alloc(v_j)) \times cost(v_j)\right),$$

in which we sum the time-cost area of each task, that is its execution time multiplied by its overestimated monetary cost. We then average the obtained value over the allowed budget $B'$. $B' \leq B^i$ is the maximum budget that any task can use in order to run. It is different from the maximum budget for the whole allocation, $B^i$, which we will use as the stop condition for the allocation algorithm.

Overestimating the costs this way allows us to guarantee that the produced allocation will not exceed the allowed budget. However, it may have a bad impact on makespan depending on how much spare time is lost. Consider a simple example to illustrate this. We want to build an allocation for a chain of 10 tasks with a budget of 10 units. One hour on a virtual instance costs 1 unit. Unfortunately each task runs for only ten minutes. With the above formula, each task will be allocated only one virtual instance as the budget limit is already reached. However, it is likely that, once scheduled, all the tasks will reuse the same instance for a total running time of 100 minutes and a cost of two units! A tighter estimation of the cost may have allowed each task to run for five minutes on two virtual CPUs, leading to a makespan divided by two for the same cost.

To hinder the effect of this overestimation, we can assume that the spare time left by each task has one in two chance to be reused by another task. The risk inherent to such an assumption is that we do not anymore have a strong guarantee that the resulting allocation will fall short of the allowed budget once scheduled. Nevertheless, we modify the definition of $cost(v_i)$ as follows:

$$cost(v_i) = \frac{\lceil T(v_i, Alloc(v_i)) \rceil + T(v_i, Alloc(v_i))}{2} \times \sum_{vm_j \in Alloc(v_i)} cost_j.$$

The definition of $T_A^{over}$ remains unchanged. However, in the remaining of this paper, it relies on this second definition of $cost(v_i)$.

Based on this definition, we propose a first allocation procedure detailed by Algorithm 2. This procedure determine one allocation for each task in the considered sub-workflow while trying to find a good compromise between the length of the critical path (hence the completion time) and the average time-cost area as defined by $T_A^{over}$.

Since the purpose of this algorithm is to determine only one allocation, we cannot simply iterate $B'$ from 0 to $B^i$. We need to estimate the value of $B'$ such that the values of $T_A^{over}$ and $T_{CP}$ will reach a tradeoff at the end of the allocation.

At convergence time, the two values are equal. $B'$ is the maximum cost of running any single task at convergence time and $B^i$ is the total cost of the allocation. As a heuristic to determin $B'$ we assume that the proportion between the total work area and the maximum work area is constant. We can therefore calculate these areas for an initial iteration and determin the value of $B'$ when convergence occurs.

$$\frac{B'}{B^i} = \frac{\sum_{j=1}^{|\mathcal{V}^i|} \left( T(v_j, Alloc^{init}(v_j)) \times cost^{init}(v_j) \right)}{T_{CP}^{init} \times \sum_{j=1}^{|\mathcal{V}^i|} cost^{init}(v_j)}$$

$Alloc^{init}$ represents the initial allocation in which we give an instance of the smallest type to every task.

Each task's allocation set is initialized with the number of CPUs of the smallest virtual instance in the catalog. Then, we determine which task belonging to the critical path would benefit the most from an extra virtual CPU, and increase the allocation of this task. We iterate this process until we find a compromise

**Algorithm 2** Eager-allocate($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), B^i$)

```
1: for all $v \in \mathcal{V}^i$ do
2:     $Alloc(v) \leftarrow \{\min_{vm_i \in \mathcal{C}} CPU_i\}$
3: end for
4: Compute $B'$
5: while $T_{CP} > T_A^{over} \cap \sum_{j=1}^{|\mathcal{V}^i|} cost(v_j) \leq B^i$ do
6:     for all $v_i \in$ Critical Path do
7:         Determine $Alloc'(v_i)$ such that $p'(v_i) = p(v_i) + 1$
8:         $Gain(v_i) \leftarrow \frac{T(v_i, Alloc(v_i))}{p(v_i)} - \frac{T(v_i, Alloc'(v_i))}{p'(v_i)}$
9:     end for
10:    Select $v$ such that $Gain(v)$ is maximal
11:    $Alloc(v) \leftarrow Alloc'(v)$
12:    Update $T_A^{over}$ and $T_{CP}$
13: end while
```

between makespan reduction and estimated cost increase. Note that the determination of $Alloc'(v_i)$ (line 7) may mean either adding a new instance with one virtual CPU to the set of resource instances already composing the allocation, or switching to another type of instance from the catalog.

Figure 7 shows an evolution of the values of $T_A^{over}$ and $T_{CP}$ across the allocation process, for a budget limit of 10 units. We have used a resource catalog inspired by Amazon EC2's catalog, which can be found in Table 1. There is a single point of convergence between the two, which represents a good trade-off between the two values. The allocation process stops if this point is reached or if the estimated costs of the allocation excedes the budget limit. In the current example, a trade-off is reached after 57 iterations.

In practice it is only worth continuing the allocation process if the value if $T_{CP}$ continues to decrease. We have added a suplimentary stop condition that is triggered if the value of $T_{CP}$ does not decrease more than one second. We call this the $T_{CP}$ *cut-off*.

As this first procedure may produce allocations that do not respect the budget constraint, we propose an alternate approach based on a similar principle as that used by the biCPA algorithm [8]. Instead of just considering the allocation that is eventually obtained when the trade-off between the length of the critical path and the average cost is reached, we keep track of intermediate allocations build as if the allowed budget was smaller. Once all these candidate allocations are determined, we build a schedule for each of them on a dedicated platform to obtain a precise estimation of their makespan they achieve and at which cost. Then it is possible to choose the "best" allocation that leads to the smallest makespan for the allowed budget.

In this second procedure, we can rely on a tighter definition of the average time-cost area that does not take spare time into account. Indeed, if some spare time exists, it will be reused (or not) when the schedule is built. Since we select the final allocation based on the resulting scheduling, we do not have to consider spare time in the first step. To some extent, it amounts to underestimate the cost of the execution of a task. Our second allocation procedure will then rely on $T_A^{under}$, defined as

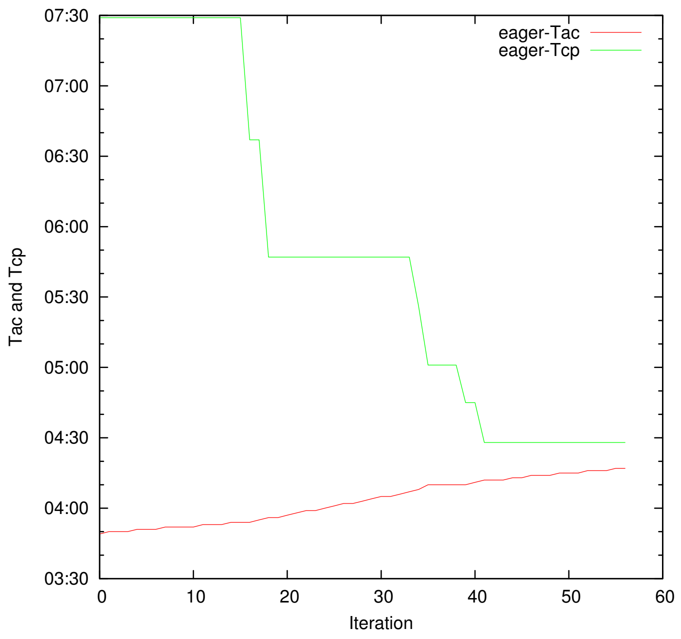


Figure 7: The evolution of $T_A^{over}$ and $T_{CP}$

.

$$T_A^{under} = \frac{1}{B'} \times \sum_{j=1}^{|\mathcal{V}^i|} (T(v_j, Alloc(v_j)) \times cost_{under}(v_j))$$

This definition differs from that of $T_A^{over}$ by the use of

$$cost_{under}(v_j) = T(v_j, Alloc(v_j)) \times \sum_{vm_k \in Alloc(v_j)} cost_k$$

that includes the exact estimation of execution time of $v_j$ and of a new variable $B'$ instead of the allowed budget $B^i$. This parameter allows us to mimic the variable size of the cluster used by the biCPA algorithm, and represents the maximum budget allowed to determine any one task's allocation. Its value will grow along with the allocation procedure, starting from the largest cost of running any task fron the initial allocation and up to $B^i$. The use of $B'$ has a direct impact on the computation of the average time-cost aera and will lead to several intermediate trade-offs and corresponding allocations. We refer the reader to [8] for the motivations and benefits of this approach.

This second allocation procedure is detailed in Algorithm 3. The first difference is on lines 5 and 20 where we determine and update the value of $B'$ to be the maximum cost of running any one task. The main difference with our first allocation procedure lies in the outer while loop (lines 6-22). This loop is used to set the value of $T_A^{under}$ that will be used in the inner loop (lines 8-16). This inner loop actually corresponds to an interval of iterations of our first allocation procedure. Each time $T_{CP} \leq T_A^{under}$ , the current allocation is stored for each

**Algorithm 3** Deferred-allocate($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), B^i$)

```
1: for all $v \in \mathcal{V}^i$ do
2:     $Alloc(v) \leftarrow \{\min_{vm_i \in C} CPU_i\}$
3: end for
4: $k \leftarrow 0$
5: $B' \leftarrow \max_{v \in \mathcal{V}^i} cost_{under}(v)$
6: while $B' \leq B^i$ do
7:     $T_A^{under} = \frac{1}{B'} \times \sum_{j=1}^{|\mathcal{V}^i|} (T(v_j, Alloc(v_j)) \times cost_{under}(v_j))$
8:     while $T_{CP} > T_A^{under}$ do
9:         for all $v_i \in$ Critical Path do
10:            Determine $Alloc'(v_i)$ such that $p'(v_i) = p(v_i) + 1$
11:            $Gain(v_i) \leftarrow \frac{T(v_i, Alloc(v_i))}{p(v_i)} - \frac{T(v_i, Alloc'(v_i))}{p'(v_i)}$
12:        end for
13:        Select $v$ such that $Gain(v)$ is maximal
14:        $Alloc(v) \leftarrow Alloc'(v)$
15:        Update $T_A^{under}$ and $T_{CP}$
16:    end while
17:    for all $v \in \mathcal{V}^i$ do
18:        Store $Allocs^i(k, v) \leftarrow Alloc(v)$
19:    end for
20:    $B' \leftarrow \max_{v \in \mathcal{V}^i} cost_{under}(v)$
21:    $k \leftarrow k + 1$
22: end while
```

task (lines 17-19), and the current allowed budget is updated (line 20). At the end of this procedure, several candidate allocations are associated with each task in the PTG.

Figure 8 shows an evolution of the values of $T_A^{under}$ and $T_{CP}$ across the allocation process, for a budget limit of 10 units. In contrast to Figure 7, here we have multiple points of convergence for the two values, each of these points represents a valid allocation with a good trade-off between the two. Since in this algorithm we underestimate the cost, there will be a lot more iterations than in the previous. The ridges in the values of $T_A^{under}$ are caused by the difference in price per CPU of the virtual machines from the catalog. As a virtual machine has more CPUs, it's price per hour decreases and so does the value of $T_A^{under}$.

It is worth noting that the value of $T_{CP}$ becomes more and more flat since the tasks' parallelism starts to become saturated. Here too we have used the $T_{CP}$ *cut-off* strategy in practice.

In a second step, we have to get an estimation of the makespan and total cost that can be achieved with each of these allocations. To obtain these performance indicators, we rely on a classical list scheduling function as shown by Algorithm 4. Tasks are considered by decreasing bottom-level values, *i.e.,* their distance in terms of execution time to the end of the application. For each task, we convert an allocation, *i.e.,* a resource request, into a mapping. This amounts to finding out which set of resource instances the task will be executed on. Two objectives have to be met. First we have to minimize the finish time of the scheduled task. Second, we have to favor reuse of spare time to reduce the schedule's cost.

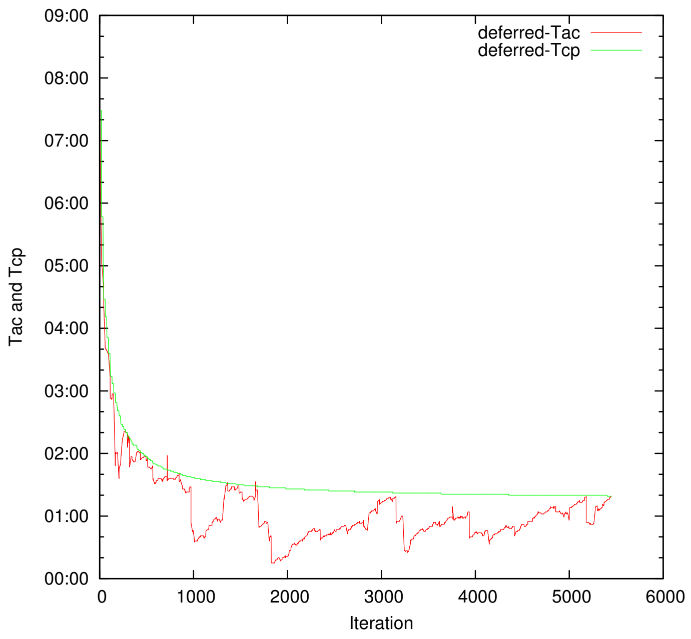


Figure 8: The evolution of $T_A^{under}$ and $T_{CP}$

.

To achieve both objectives, we proceed in two steps. First, we estimate the finish time a task will experience by launching only new instances to satisfy its resource request. This set of newly started instances is built so that its cost is minimum, *i.e.,* favor big and cheap instances from the catalog. However, we don't make any assumption about spare time reuse for this mapping. Hence, its cost is computed by rounding up the execution time of the task. This provides us a baseline both in terms of makespan and cost for the current task. Second, we consider all the already started instances, *i.e.,* launched by already scheduled tasks, to see if some spare time can be reused and thus save money. We sort these instances by decreasing amount of spare time (from the current time) and then by decreasing size. Then we select instances from this list in a greedy way until the allocation request is fulfilled, and estimate the finish time of the task on this allocation, as well as the cost of it. This cost is computed as the product of the rounded up execution time of the task by the cost of each instance used minus the cost of the reused spare time.

Now, we have two possible mappings for the current task with different finish times and costs. Our algorithm selects the candidate that leads to the earliest finish time for the task. If the two mappings lead to the same finish time, we select the cheapest option. This is summarized in Algorithm 4.

At the end of a call to Algorithm 4, we have an estimation of the makespan and total cost of the schedule of $\mathcal{G}^i$ using a given allocation. This algorithm is called for each $Allocsi(k, *)$ as determined by Algorithm 3.

Algorithm 5 details the three stages of our second allocation procedure: (i) Determine a set of candidate allocations for each task (lines 1-3 and Algo-

**Algorithm 4** List-schedule($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), Allocs(*) = Allocs^i(j, *)$)

```
1: running_instances ← $\emptyset$
2: for all $v \in \mathcal{V}^i$ in decreasing order of bottom-level values do
3:     $new$ ← cheapest set of new instances that fulfill $Allocs(v)$
4:     $cost(new) = \lceil T(v, Allocs(v)) \rceil \times \sum_{vm_j \in new} cost_j$
5:     $finish(new)$ ← finish time of $v$ on $new$
6:     Sort all $vm_j \in$ running_instances by decreasing spare time and size
7:     $reuse$ ← first set of instances from running_instances that fulfill $Allocsv)$
8:     $cost(reuse) = (\lceil T(v, Allocsv)) \rceil -$ reused spare time$) \times \sum_{vm_j \in reuse} cost_j$
9:     $finish(reuse)$ ← finish time of $v$ on $reuse$
10:    if $finish(reuse) < finish(new)$ then
11:        $map(v) \leftarrow reuse$
12:    else if $cost(new) < cost(reuse)$ then
13:        $map(v) \leftarrow new$
14:    else
15:        $map(v) \leftarrow reuse$
16:    end if
17:    running_instances ← running_instances $\cup\, map(v)$
18: end for
19: $cost \leftarrow \sum_{vm_j \in VMs} \lceil T_{end_j} - T_{start_j} \rceil \times cost_j$
20: $makespan \leftarrow \max(T_{end_j}) - \min(T_{start_k}), \forall vm_j, vm_k \in$ running_instances
21: return $(makespan, cost)$
```

rithm 3); (ii) Compute the respective makespans and costs achieved by mapping each allocation on a dedicated IaaS cloud (line 7 and Algorithm 4); and (iii) Select the allocation that leads to the best makespan while respecting the budget constraint based on the couples returned by Algorithm 4

## 4.4 Scheduling and workflow execution

It is worth noting that all the previous steps are all static and are performed before runtime. Currently we do not address the problem of workflow execution, as it is not possible to take into consideration the possible state of the Cloud platform and therefore, the resulting schedule would be based on false information. However, by using the allocations selected by our approach we can guarantee that the initial workflow will be run on the Cloud platform given the inital budget, with a certain confidence.

When constructing a schedule by starting from the chosen allocations one should take into consideration the following points: a) as a result of non-determinism, two or more sub-workflows can be ready for scheduling at the same time, yet it is not trivial to find the best order in which they should be scheduled; b) if scheduling is performed offline, there is no possible way of knowing the state of the platform and therefore it is highly likely that the estimations used while scheduling would be false.

**Algorithm 5** Find-allocations($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), B^i$)

```
1: for all $v_j \in \mathcal{V}^i$ do
2:     $Allocs^i \leftarrow Deferred\text{-}allocate(\mathcal{G}^i, B^i)$
3: end for
4: $selected\_allocation \leftarrow \emptyset$
5: $best\_makespan \leftarrow +\infty$
6: for all $Allocs^i(k,*) \in Allocs^i$ do
7:     $(makespan, cost) \leftarrow$ List-schedule($\mathcal{G}^i, Allocs^i(k,*)$)
8:     if $(makespan < best\_makespan) \wedge (cost \leq B^i)$ then
9:         $best\_makespan \leftarrow makespan$
10:        $selected\_allocation \leftarrow Allocs^i(k,*)$
11:    end if
12: end for
```

## 5 Experimental evaluation

### 5.1 Experimental methodology

We use simulations with synthetic PTGs to evaluate our claims. The synthetic PTGs were generated based on three application models: Fast Fourier Transform (FFT), Strassen matrix multiplication and random workloads that allow us to explore a wider range of possible applications. For more details related to the synthetic workloads and their generation we would like to reffer the reader to [8], section V.

### 5.2 Platform description

Throughout our experiments we have used Amazon EC2 as our model IaaS platform. This is visible in the virtual resource catalog that we have used, inspired by the the available virtual resource instance types of Amazon EC2 [4] and described in Table 1.

| Name | #VCPUs | Network performance | Cost / hour |
|---|---|---|---|
| m1.small | 1 | *moderate* | 0.09 |
| m1.med | 2 | *moderate* | 0.18 |
| m1.large | 4 | *high* | 0.36 |
| m1.xlarge | 8 | *high* | 0.72 |
| m2.xlarge | 6.5 | *moderate* | 0.506 |
| m2.2xlarge | 13 | *high* | 1.012 |
| m2.4xlarge | 26 | *high* | 2.024 |
| c1.med | 5 | *moderate* | 0.186 |
| c1.xlarge | 20 | *high* | 0.744 |
| cc1.4xlarge | 33.5 | 10 Gigabit Ethernet | 0.186 |
| cc2.8xlarge | 88 | 10 Gigabit Ethernet | 0.744 |

Table 1: Amazon EC2's virtual resource types

In our catalog we did not consider instances of type *t1.micro* as it receives virtual CPUs in bursts, which makes it difficult to quantify. We also did not consider GPU cluster instances (*cg1.4xlarge*) as their GPU resources are difficult to quantify in virtual CPUs.

Given that the network bandwidth information for the *m1*, *m2* and *c1* type instances is not given, we have considered *high* network performance as being 10 Gigabit Ethernet and *moderate* network performance as being 1 Gigabit Ethernet.

## 5.3 Comparison of running times

We can consider the running time of the two allocation algorithm on a 16-core Intel Xeon CPU running at 2.93GHz. For convenience's sake we have considered the running time of Eager relative to Deferred for the same PTG and budget. A plot of the relative running time across all the simulation scenarios for each type of application can be seen in Figure 9. The first quartile has 25% of the total values smaller or equal to it, the second quartile (median) has 50% and the third quartile has 75%. The range between the first and third quartile is the inter-quartile range (IQR). The whiskers of the plot extend from the ends of the box to 1.5 times the IQR. For convenience's sake, outliers are not show.

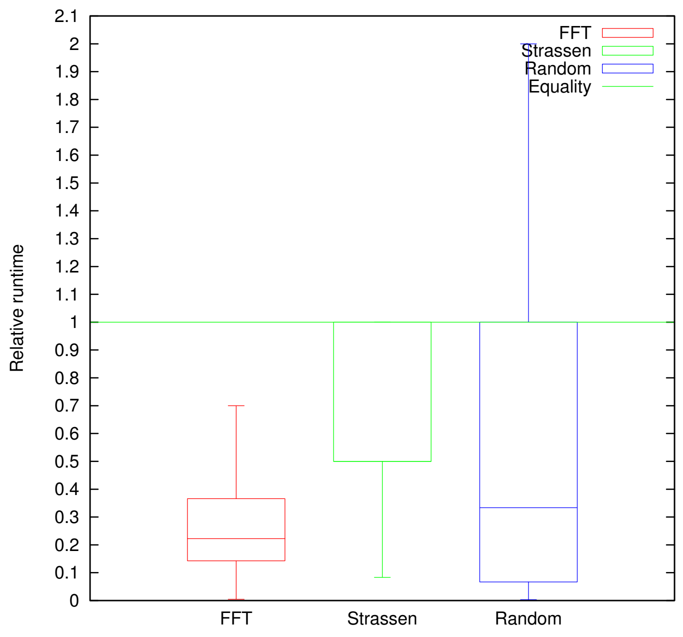


Figure 9: Relative runtime of the two allocation algorithms $\left(\frac{Time\ to\ compute\ Eager}{Time\ to\ compute\ Deferred}\right)$

Deferred's outside iteration over the budget limit has a visible influence, especially for higher values of the maximum budget. Deferred's running time is slower than Eager's by at most an order of magnitude. It is worth noticing that the behaviour is as expected, Eager is significantly faster than Deferred for almost all the allocations performed. In the situation of small PTGs, both algorithms run considerably fast and in these situations, the resolution of the internal clock can introduce disturbances, as seen in the case of random PTGs.

## 5.4 Simulation results

We have varied the budget limit for all the input PTGs from 1 unit to 50 units. By considering the cost per hour of the cheapest VM type (0.0084 per CPU per hour) from the catalog in Table 1 gives a testing interval from a mimimum of 11 CPU hours to a maximum of 5914 CPU hours. This has the double role of permitting bigger PTG to manifest their influence over time to produce a more general trend and stressing the algorithms in order to find out their best operating parameters.

Figures 10, 11, 12 and 13 shows plots of aggregated results of makespan and cost after task mapping, for all three application types. We have used the same semantics for quartiles and whiskers as previously explained.

The first observation worth noting is that up to a certain budget value Eager passes the budget limit. This means that our initial assumption of 50% VM spare time reuse is an optimistic one. After a certain budget limit, Eager reaches a point of saturation due to the $T_{CP}$ *cut-off* strategy. This means that after a certain budget limit, the same allocation will be produced by Eager and, consequently, the same task mapping after scheduling.

While the $T_{CP}$ *cut-off* strategy also applies to Deferred, it does not try to estimate the costs, it always underestimates them while performing allocations. As a result, the actual costs of the allocations given by Deferred will be a lot higher than the budget limit and the actual saturation level will also be higher. As expected, Deferred in combination with Algorithm 5 will always select an allocation that, after task mapping, is within the budget limit. In combination with a high saturation level this, yields the behavior that we see in Figure 11. The only moment when Deferred produced allocations that are not in the budget limit is when the budget limit is too low to accomodate all the tasks in the workflow.

To ease the comparison between the two approaches, we can consider the plots in Figures 14 and 15. It can be seen that, in the beginning, the makespans produced by Eager allocations are shorter than those produced by Deferred allocations and from a cost point of view, Eager produces more costly allocations than Deferred. As the budget increases, the balance shifts slightly in favour of Eager for cost and Deferred for makespan, yet it is not as unbalanced as in the beginning.

For small values of the budget *i.e.,* before task parallelism starts to become saturated, Eager outperforms Deferred in terms of resulting makespan by a median of as much as 12%, but Deferred never passes the budget limit and outperforms Eager in terms of budget by a median of as much as 26%. The situation changes once task parallelism begins to appear and the two algorithms yield the same makespan with a median difference of 2%, yet Eager outperforms Deferred in terms of cost by as much as 23%. It it therefore intuitive that for small applications and small budget values one should use Deferred, but when the size of the applications increases significantly or the budget limit approaches task parallelism saturation, using Eager would be the best strategy.

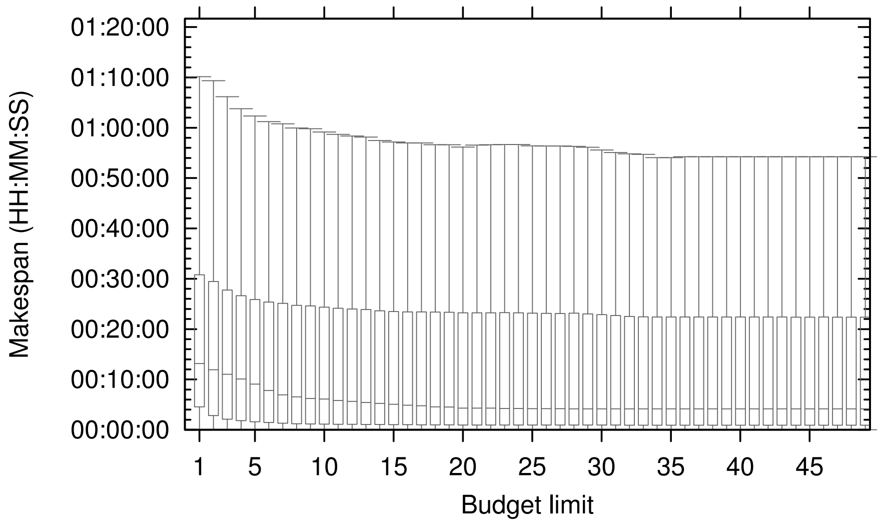


Figure 10: Makespan using Eager allocation using all workflow applications

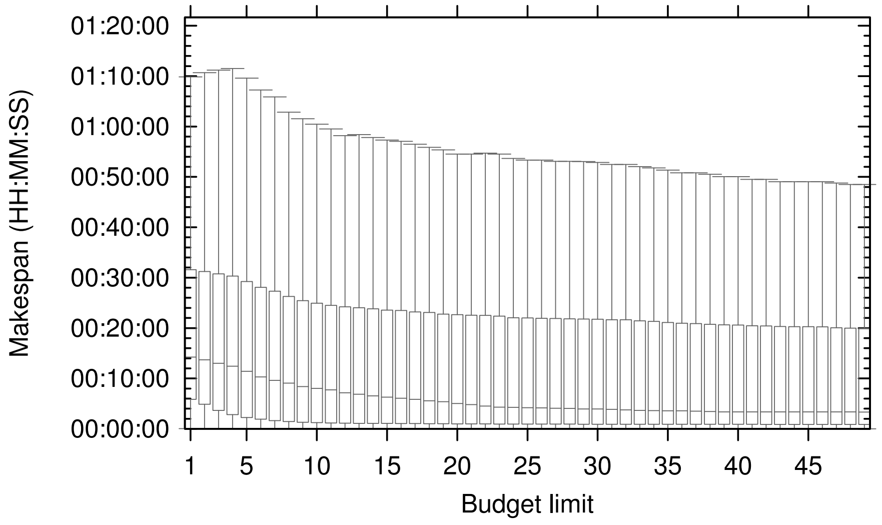


Figure 11: Makespan using Deferred allocation using all workflow applications

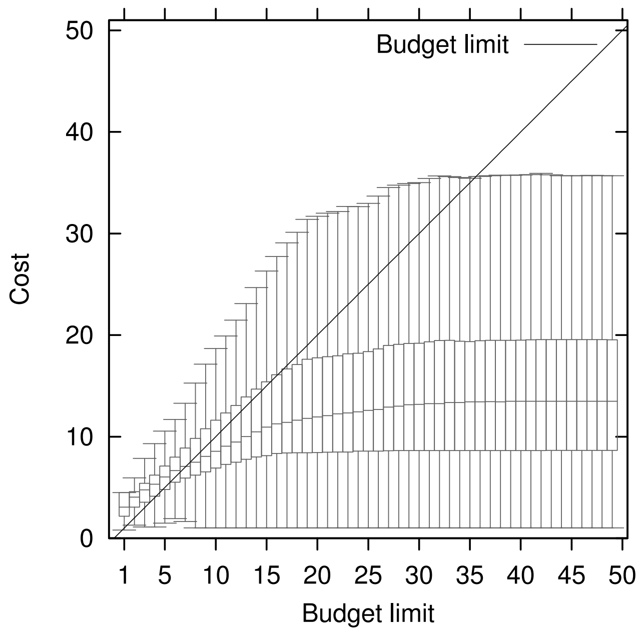


Figure 12: Cost using Eager allocation using all workflow applications

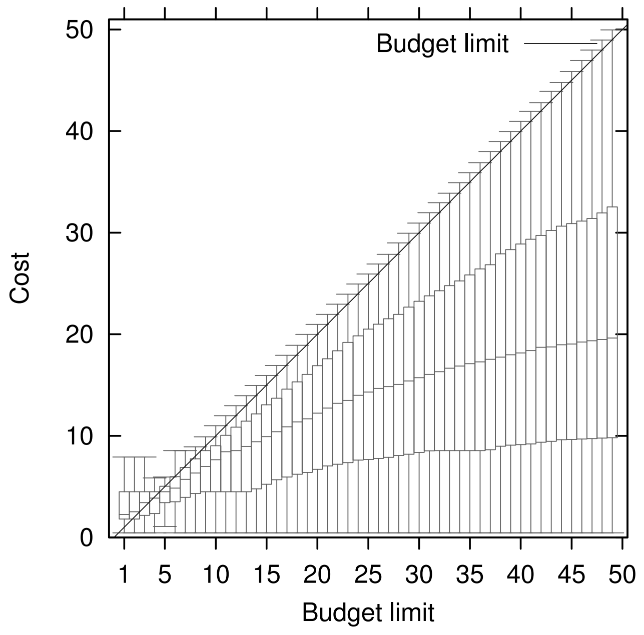


Figure 13: Cost using Deferred allocation using all workflow applications

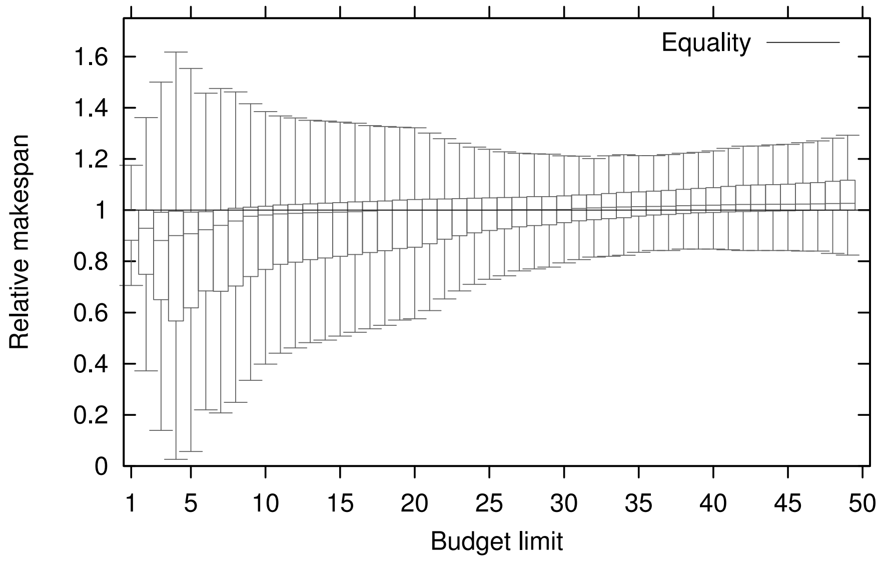


Figure 14: Relative makespan ($\frac{Eager}{Deferred}$) for all workflow applications

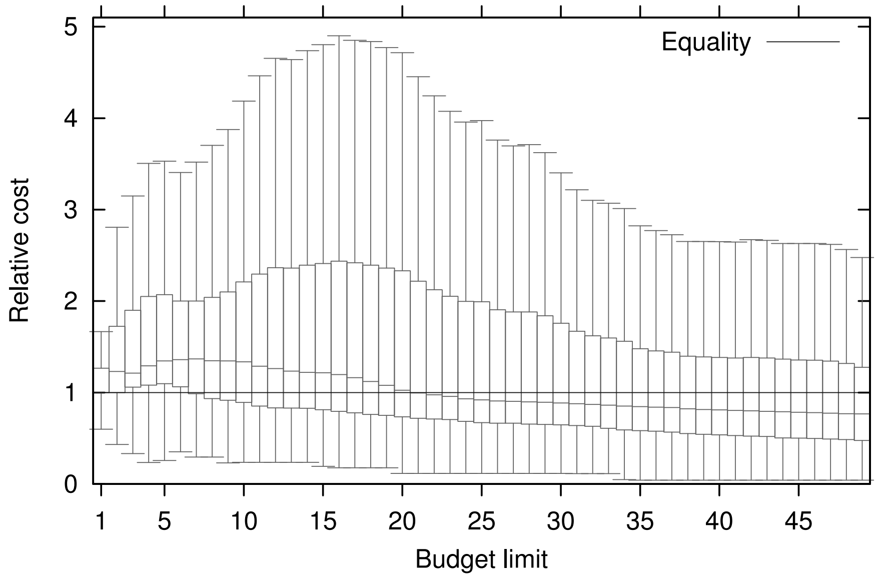


Figure 15: Relative cost ($\frac{Eager}{Deferred}$) for all workflow applications

## 6 Conclusion and Future Work

The elastic allocations that Cloud platforms offer has opened the way for more flexible data models. Notably, parallel task graph applications with a more complex structure than classic DAG workflows are a good match for the elastic allocation model. There has been lots of work around the topic of parallel task graph scheduling on grid or Cloud platforms, yet none of the previous approaches focus on both elastic allocations and non-DAG workflows.

In the current article we present our research on the topic of scheduling with budget constraints for non-DAG workflow models that target Cloud platforms. Our approach is to transform the original problem into a set of smaller sub-problems that have been studied before and propose a solution for them. Concretely, we split the input non-DAG workflow into DAG sub-workflows. Next we present two allocation algorithms, Eager and Deferred, built on the specifics of a typical IaaS Cloud platform and provide an algorithm for selecting the most interesting of these allocations such that the budget limit is not reached. Eager is designed to be a fast allocation algorithm and uses a heuristic approach for estimating the real cost of the allocation it produces. Deferred, on the other hand, is slower in running time, but it produces a set of allocations, each with a good trade-off between the time on the critical path and the total work area (in cost). It does not try to estimate the real cost of the allocations, but underestimates it instead and delayes the decision of which allocation to choose until scheduling time. The two algorithms differ in terms of running time by as much as an order of magnitude in favour of Eager. Under tight budget constraints, Eager leads to shorter, yet more expensive schedules and usually passes the budget limit. In contrast, Deferred always results in schedules that are in the budget limit and longer as makespan. The conclusion is that for small applications or small budget limit sizes, Deferred yields the best results and for large applications or large budget limit sizes Eager outperforms Deferred.

As long term goal we plan on integrating the current work into an existing Open Source IaaS Cloud platform. A good improvement will be to determine per application which is the tipping point upto which Deferred should be used and after which Eager would be the best fit.

## Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

## References

[1] van der Aalst, W., Barros, A., ter Hofstede, A., Kiepuszewski, B.: Advanced Workflow Patterns. In: Proc. of the 7th Intl. Conference on Cooperative Information Systems. pp. 18–29 (2000)

[2] Altintas, I., Bhagwanani, S., Buttler, D., Chandra, S., Cheng, Z., Coleman, M., Critchlow, T., Gupta, A., Han, W., Liu, L., Ludäscher, B., Pu,

C., Moore, R., Shoshani, A., Vouk, M.A.: A modeling and execution environment for distributed scientific workflows. In: Proc. of the 15th Intl. Conference on Scientific and Statistical Database Management. pp. 247–250 (2003)

[3] Amazon Elastic Compute Cloud (Amazon EC2): `http://aws.amazon.com/ec2` (2012)

[4] Amazon Elastic Compute Cloud (Amazon EC2): `http://aws.amazon.com/ec2/instance-types/` (2012)

[5] Bahsi, E.M., Ceyhan, E., Kosar, T.: Conditional Workflow Management: A Survey and Analysis. Scientific Programming 15(4), 283–297 (2007)

[6] Couvares, P., Kosar, T., Roy, A., Weber, J., Wenger, K.: Workflow Management in Condor. In: Taylor, I., Deelman, E., Gannon, D., Shields, M. (eds.) Workflows for e-Science, pp. 357–375. Springer (2007)

[7] Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J., Katz, D.: Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. Scientific Programming Journal 13(3), 219–237 (2005)

[8] Desprez, F., Suter, F.: A Bi-Criteria Algorithm for Scheduling Parallel Task Graphs on Clusters. In: Proc. of the 10th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing. pp. 243–252 (2010)

[9] Fahringer, T., Jugravu, A., Pllana, S., Prodan, R., Jr., C.S., Truong, H.L.: ASKALON: a Tool Set for Cluster and Grid Computing. Concurrency - Practice and Experience 17(2-4), 143–169 (2005)

[10] Fernandez, H., Tedeschi, C., Priol, T.: A Chemistry Inspired Workflow Management System for Scientific Applications in Clouds. In: Proc. of the 7th Intl. Conference on E-Science. pp. 39–46 (2011)

[11] Glatard, T., Montagnat, J., Lingrand, D., Pennec, X.: Flexible and Efficient Workflow Deployment of Data-Intensive Applications on GRIDS with MOTEUR. Intl. Journal of High Performance Computing Applications 3(22), 347–360 (2008), special issue on Workflow Systems in Grid Environments

[12] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice and Experience 18(10), 1039–1065 (2006)

[13] Mao, M., Humphrey, M.: Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In: Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, (SC'11) (2011)

[14] Mayer, A., McGough, S., Furmento, N., Lee, W., Newhouse, S., Darlington, J.: ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In: UK e-Science All Hands Meeting. pp. 627–634. IOP Publishing Ltd (2003)

[15] Oinn, T., Greenwood, M., Addis, M., Alpdemir, N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M., Senger, M., Stevens, R., Wipat, A., Wroe, C.: Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. Concurrency and Computation: Practice and Experience 18(10), 1067–1100 (2006)

[16] Radulescu, A., van Gemund, A.: A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In: Proc. of the 15th Intl. Conference on Parallel Processing (ICPP). pp. 69–76 (2001)

[17] Taylor, I., Shields, M., Wang, I., Harrison, A.: The Triana Workflow Environment: Architecture and Applications. Workflows for e-Science pp. 320–339 (2007)

[18] Viroli, M., Zambonelli, F.: A Biochemical Approach to Adaptive Service Ecosystems. Information Sciences 180(10), 1876–1892 (2010)

[19] Werner, T.: Target Gene Identification from Expression Array Data by Promoter Analysis. Biomolecular Engineering 17(3), 87–94 (2001)

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