Minimizing rental cost for multiple recipe applications in the Cloud

Fouad Hanna[*], Loris Marchal[†], Jean-Marc Nicod[*], Laurent Philippe[*], Veronika Rehn-Sonigo[*] and Hala Sabbah[*]

Abstract

Clouds are more and more becoming a credible alternative to parallel dedicated resources. The pay-per-use pricing policy however highlights the real cost of computing applications. This new criterion, the cost, must then be assessed when scheduling an application in addition to more traditional ones as the completion time or the execution flow. In this paper, we tackle the problem of optimizing the cost of renting computing instances to execute an application on the cloud while maintaining a desired performance (throughput). The target application is a stream application based on a DAG pattern, i.e., composed of several tasks with dependencies, and instances of the same execution task graph are continuously executed on the instances. We provide some theoretical results on the problem of optimizing the renting cost for a given throughput then propose some heuristics to solve the more complex parts of the problem, and we compare them to optimal solutions found by linear programming.

Keywords: Clouds, scheduling, optimization, DAG applications

1 Introduction

Public clouds are more and more becoming a credible alternative to private parallel computing resources to run scientific applications. The pay-per-use pricing policy however highlights the financial cost of running an application on cloud platforms while only the computing cost where mostly considered on dedicated platforms. From the scheduling point of view this introduces a new criterion, the cost, when executing an application on a cloud in addition to more traditional ones as the completion time or the execution flow. From the optimization point of view, the cost criterion is antagonistic to the execution performance both in case of an execution flow or a single application as it is cheaper not to run an application than running it.

In this paper we tackle the problem of optimizing the cost of renting computing instances to execute an application on the cloud while maintaining a desired performance (throughput). The target application model is a stream of applications based on the same DAG pattern, i.e., composed of several tasks with dependencies, and instances of the same execution task graph are continuously executed on the instances. An example of such an application can be found in image or signal processing applications where several different filters or codecs must be applied to a stream of data, or a large set of images. Our objective is to provide enough resources for the application to reach a given throughput, in terms of computed instances, while minimizing the renting cost. The desired throughput may be mandatory to guarantee the quality level of a video stream for example. Since we are planning to rent machines from clouds, we are interested in minimizing the total cost. As we run long-term stream applications, this objective is better expressed as minimizing the hourly price for renting the machine.

On the cloud the provided resources are however heterogeneous as different types of instances are proposed: for instance the EC2 or Azure clouds provide different types of on demand instances[*]. Other heterogeneity factors must also be taken into account: 32-bit architectures vs. 64 ones, CPU architectures vs. GPU ones and so on. This imposes constraints on the task to instance mapping: tasks that can run on one instance cannot always run on another instance. For instance 64-bits applications cannot be run on 32-bits instances, CPU tasks cannot be run on GPU instances or some constraints may be forced by the amount of memory required to run the task. On the other hand tasks may have different implementations with different constraints, as

[*]FEMTO-ST Institut, Univ. Bourgogne Franche-Comté/CNRS, France

[†]LIP, ENS Lyon/CNRS/INRIA, France

[*]https://aws.amazon.com/ec2/instance-types

for instance the need for specific hardware. A matrix multiplication task can be implemented either for CPU or for GPU but the associated task must be run on the right hardware. So a type must be associated with each task and only the corresponding instances can be used to run it. Considering that the same computation can be done using tasks of different types, we can define alternative graphs to compute the same result. For instance, if an application pattern includes a matrix multiplication task then two alternative graphs may be defined: one that uses a CPU task to compute the matrix multiplication task and another that uses a GPU task to compute it.

As our objective is only to reach a given throughput we can run several alternative graphs concurrently. All the graphs participate to the same application as long as the global cost is lowered. Considering that we can access clouds that provide different instance types (CPU speed, memory, I/O bandwidth, GPU or not, etc.) at different prices we try to optimize their use, and their cost, to reach a target throughput. The graphs may share or not the rented cloud instances. This makes the problem more complex than with a single application graph. Thus, the problem is on the one hand to find the right application graph(s) that optimize the cost, and on the other hand the right cloud instances that gives the lower cost with this or these application(s).

Note that because of the use of various application graphs, different instances, such as different images of a video stream, may be processed using different graphs, and thus experience different processing times. To avoid that these images are output in a different order than their input order, a buffer of sufficient size is needed. In this paper, we assume that we have such a buffer as well as a mechanism that ensures that instances are output in the same order as their input order, and we concentrate only on the throughput maximization.

In the following, we study this problem using various models of the application programs. In section 2 we present the related work on cloud scheduling and cost optimization. The studied model is presented in section 3 then we provide some theoretical results on the problem of optimizing the cost for a given throughput in section 4 and 5. In section 6 we propose some heuristics to solve the more complex parts of the problem and we compare them to optimal solutions found by linear programming in section 8.

2 Related work and context

We focus in this paper on streaming applications run on heterogeneous resources and in particular on *coarse grain* applications. These applications are composed of several tasks, linked by dependency constraints, and they continuously process a set of input data to compute the output set. In fact, this computation may be truly continuous as in the case of multimedia or sensor applications [10, 11, 14] or long enough to neglect the initialization and ending phases and concentrate on the steady-state phase as in [2]. Much attention has already been paid to running workflow applications on heterogeneous resources as a grid. These studies may be distinguished in two main categories: practical works or more theoretical ones. From the practical point of view several frameworks as Pegasus [6] or Condor [16] propose to dynamically schedule multi-task workflows on heterogeneous computing resources. For example, a survey on scheduling pipelined workflow applications on grids is given in [3]. Several objectives are studied in this context depending on the workflow characteristics. When the executed workflows are different, the makespan objective is usually targeted [15] but when the same instance of the workflow is continuously run, the throughput objective is more adapted. From the theoretical point of view several works tackle the problem of proving complexity results for throughput. In [2] the optimal throughput is given by defining a periodic schedule but other objectives, the latency [5] or the reliability [4], may also be added to improve the quality of the result. Note that the cost objective is not truly relevant in the grid context as the resources are usually freely shared and there is no economical model behind.

By providing on-line on-demand resources, the cloud concept goes one step further in the similarity to flexibility to the Power Grid, the so-called elasticity. Clouds are traditionally classified depending on their public opening, i.e., public or private clouds, or depending on the service level [1, 8], i.e., Infrastructure as a service or IaaS, Plateform as a service or PaaS, Software as a service or SaaS, and so on. As we are interested in executing applications by using on-demand resources in the cloud we focus in the following on pubic IaaS clouds. The Amazon EC2[†] or the Microsoft Azure cloud[‡] are examples of such clouds.

IaaS public clouds introduce an economical di-

[†] `http://Aws.Amazon.Com/Ec2/`
[‡] `http://www.windowsazure.com/`

mension in the resource use with a market oriented model. The pay-as-you-go pricing model makes it worth to study the application cost before execution. In the context of application execution optimization in cloud computing, the cost objective is mainly studied from a practical or dynamic point of view. For instance [17] studies the impact of pricing on distributed applications, [13] tackles the problem of minimizing the cost of cloud use, [18] optimizes the task to virtual machine assignment based on QoS requirements, and [19] addresses the question of matching customer demand and provider's revenues. In [9] several strategies of dynamic resource provisioning of homogeneous instances are assessed to lower the cost and wait objectives with an online model and in [7] scheduling heuristics for workflow applications are compared on real heterogeneous instances. As far as we know there is no work on minimizing the execution cost of a workflow application processing a stream of data on the cloud.

3 Framework

In this section we formally define the platform and application framework. This model is rather traditional with a set of tasks to be run on processing resources, cloud instances here. The problem we face is to provision enough resources to run a DAG based streaming application with a given throughput. The cloud provides heterogeneous resources, different machines with different throughputs and different costs. We want to find the cheapest configuration that allow to reach the desired throughput.

As the considered tasks are typed, only a part of the provided resources can be used to run one of the tasks. Yet all the tasks of the same type can be run on the same instance provided that its throughput is sufficient. As stated in section 1, different application graphs may be used to compute the same input set, either because they use different algorithms or because they take benefit of different types of resources.

Note that we concentrate on computing intensive applications and thus we neglect the communications costs between the tasks as a first approximation.

The application framework consists of a *global application* ϕ, a set of J workflow applications, or graphs, where each graph φ^j ($1 \leqslant j \leqslant J$) is composed by I_j tasks φ_i^j ($\varphi^j = \{\varphi_1^j, \ldots, \varphi_i^j, \ldots, \varphi_{I_j}^j\}$). A type is associated to each task. Let $\mathcal{T} = \{1, 2, \ldots, Q\}$ be the set of Q (task) types available on the platform and t be a function that returns the type q of a task φ_i^j such that $t(i, j) = q$. An example is given in Figure 1 to illustrate these notations considering one application and four types. This application can be performed using one of the three alternative application graphs. All these application graphs have tasks of type 1. Hence a machine able to perform that type can be shared by the three application graphs. One can see that the type 3 is also shared between the two first application graphs of this illustrating example.

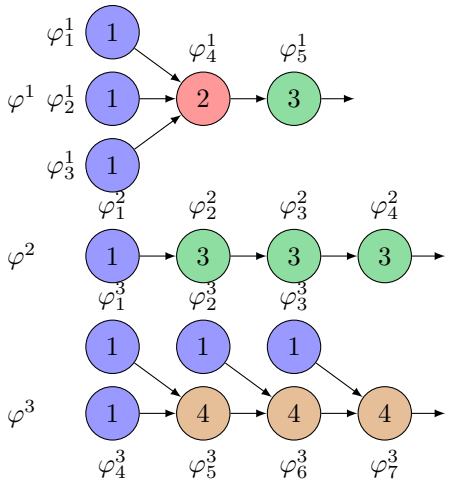

Figure 1: Illustrating task graphs

These applications are to be run on virtual machine instances, or processors, available from the cloud. To be able to potentially run a copy of each task graph on the computing resources, we need at least one processor able to process each of its task types.

We consider here dedicated resources: a task type also corresponds to a processors type, which is the type of processor capable of processing tasks of this type. Thus, a task φ_i^j of type q is only processed by a processor of the same type and it is the only task type that such a processor may process. Processors of different types have different renting costs: the cost of renting a processor of type q is denoted by c_q. Note that as we aim at providing a desired throughput for an unknown (but long) period of time, this cost is a hourly rate. The throughput of a processor of type q continuously processing tasks of the same type is r_q. Finally, we consider that all processors of the same type have the same characteristics (throughput and cost).

The Global Application ϕ requires a certain level of quality of service (QoS) which is expressed by its

output throughput ρ (number of data sets computed by time unit ($t.u.$). Since the output data sets may be computed using different workflow applications, the throughput ρ is the sum of the throughput ρ_j of each workflow application $\varphi^j \in \phi$. Each local throughput ρ_j has to be determined considering the number of resources (and possibly the fraction of them) dedicated to this workflow application.

Each application graph may include several tasks of the same type: for example on Figure 1, graph φ^3 has 4 tasks of type 1. We denote by n_q^j is the number of tasks of type q in the application graph φ_j. In the previous example, we have $n_1^3 = 4$.

Table 1 summarizes the main definitions used in the allocation problems. All the parameters describing the applications and the platform are integers.

Dimensions	
Q	number of task and processor types
J	number of graphs
I_j	number of tasks in the graph φ^j
Indices	
q	a task of processor type
j	a graph
i	a task in a given graph φ^j
Parameters	
c_q	cost of renting a processor of type q
r_q	throughput of a processor of type q
n_q^j	number of tasks of type q in the graph φ^j
Variables	
ρ	total throughput of the set of graphs
ρ_j	individual throughput of the graph φ^j
x_q	number of rented processors of type q
$C_q(\rho)$	rental cost of processor of type q to achieve ρ
$C(\rho)$	cost of the platform to achieve the throughput ρ

Table 1: Main definitions for the allocation problems

The problem we face is a dimensioning problem where we want to rent enough computing instances from (possibly different) clouds to assess the QoS requirements of the Global Application ϕ while minimizing the global rental cost C. To achieve this goal, we have to select which graphs φ^j will be used, and with which throughput ρ_j (an unused graph will simply be considered as $\rho_j = 0$). The number of necessary computing resources for each task type q has also to be determined. The general problem is expressed as follows.

Definition 1 *MinCOST Given an application described by J possible application graphs, a platform described by the unitary costs c_q, throughputs r_q for each processor of type q and a target throughput ρ, what is the throughput ρ_j of each application graph φ^j and the number of processors x_q of each type q to be booked to reach the prescribed throughput with minimal cost ?*

In the following, we propose several practical ways to find solutions for this problem, depending on the complexity of the application description.

4 Simple cases

In this section, we first focus on two simple variants of the problem, which allows to give a first overview of the complexity of the problem.

4.1 Single application graph

In the first simple case we consider one application made of a single graph φ^1: this is the only available option to produce the final result. For each task type q, we need to rent a sufficient number of processors from the Cloud. In particular, the application graph may include several tasks of the same type q. For each task type q the number of machines needed to reach the throughput ρ is given by:

$$x_q = \left\lceil \frac{n_q}{r_q} \cdot \rho \right\rceil$$

The associated cost $C_q(\rho)$ to compute the tasks of type q in the application is:

$$C_q(\rho) = \left\lceil \frac{n_q}{r_q} \cdot \rho \right\rceil \times c_q$$

Finally, the global cost is:

$$C(\rho) = \sum_{q=1}^{Q} C_q(\rho) = \sum_{q=1}^{Q} \left\lceil \frac{n_q}{r_q} \cdot \rho \right\rceil \times c_q$$

4.2 Several independent applications

The second simple case considers several independent applications $\varphi^1, \ldots, \varphi^J$. Each of these applications produces its own result. The difference with our general context is that the throughput for each application is now prescribed: application φ^j must have a throughput of at least ρ_j instances per time unit. Note that this makes the problem *simpler*, as

the decomposition of the total throughput ρ into the sum $\rho_1 + \cdots + \rho_J$ is fixed.

Since different task graphs are involved, several tasks of different graphs may have the same type. Consequently, the different graphs may share machines of the same type. Contrarily to the previous simple case, all application graphs have to be considered when computing the number of machines of type q to rent. For each task type q the number of machines needed to reach the throughput ρ is given by:

$$x_q = \left\lceil \frac{\sum_{j=1}^{J} n_q^j \times \rho_j}{r_q} \right\rceil .$$

The associated cost C_q to compute the tasks of type q in the application is:

$$C_q(\rho_1, \rho_2, \ldots, \rho_J) = \left\lceil \frac{\sum_{j=1}^{J} n_q^j \times \rho_j}{r_q} \right\rceil \times c_q$$

Finally, the global cost of the required platform is:

$$C(\rho_1, \rho_2, \ldots, \rho_J) = \sum_{q=1}^{Q} C_q(\rho) = \sum_{q=1}^{Q} \left\lceil \frac{\sum_{j=1}^{J} n_q^j \times \rho_j}{r_q} \right\rceil \times c_q$$

5 General problem with various application complexities

We now get back to the general problem where a single result has to be computed with a prescribed throughput ρ. Several application graphs φ^j can be used to compute this result. Every application produces the same result but each at its own throughput ρ_j which contributes to the global throughput ρ. In this case every individual throughput ρ_j has to be determined for each application.

To address this problem we consider the three following cases with an increasing complexity:

1. In the first case, each application graph consists of a single task, whose type is different from the types of other application tasks. In that case an application graph is seen as a simple black box, which makes it easy to compute the cost of a graph given its throughput. We aim at balancing the throughput between the applications to lower the global cost.

2. In the second case, we do not consider application graphs as black boxes anymore but as sets of tasks that do not share any task type: all task types used from application graph φ^j are specific to that graph. This case arise for example if we use several computing clouds to run the application graphs computing the same result: a graph running on a given cloud cannot share its resources with another graph running on a different cloud.

3. In the third case, we consider that the application graphs can share task types. This is the general case. As one processor can be shared by several application graphs, their throughput depend on each other: throughputs and resource sharing must be considered at the same time. As resources may be shared between several applications then only one cloud can be considered.

In the following we present solutions for these three cases.

5.1 Black box applications

We first present the simplest variation when the applications φ^j are considered as black boxes, all different from each other, or when they consist of only one task φ_1^j whose type is q ($t(1, j) = q$). Note that each task type corresponds to only one application. Thus for simplification reasons we use $j = q$. We have:

$$\forall j \text{ and } \forall j' (1 \leqslant j, j' \leqslant J) : \; t(1, j) = t(1, j') \Rightarrow j = j'$$

Let ρ_q be the throughput of application $\varphi^j = \varphi^q$. The problem can be expressed as an integer linear problem where x_q denotes the number of machines of type q used:

$$\begin{cases} \text{Minimize } C(\rho) = \displaystyle\sum_{q=1}^{Q} x_q c_q \\ \text{Under the constraint } \displaystyle\sum_{q=1}^{Q} x_q \rho_q \geqslant \rho \end{cases}$$

This resembles a knapsack problem with repetition using negative weights and values. Such a knapsack problem is formalized as follows.

Definition 2 (Unbounded Knapsack Problem) *Given n objects with value v_i and weight w_i, and a total capacity of W, how many copies of each object should we select to maximize the total value without exceeding weight W ?*

The previous knapsack problem can be expressed with the following integer linear program, where x_i is the number of copies of item i included in the solution.

$$\begin{cases} \text{Maximize } \sum x_i v_i \\ \text{Under the constraint } \sum x_i w_i \leqslant W \end{cases}$$

Our problem is thus equivalent to a knapsack problem where items have value $(-c_q)$ and weight $(-\rho_q)$ and the total capacity is $(-\rho)$. Solving such a knapsack is a (unary) NP-complete problem and many approximation algorithms and heuristics have been proposed to solve it [12]. In particular, there exists a pseudo-polynomial dynamic program which solves it with time complexity $O(n\rho)$. This solution can easily be translated into a solution to our problem for this case.

5.2 Applications without shared task types

We now consider that the global application ϕ is composed of several application graphs $\varphi^1, \ldots, \varphi^j, \ldots, \varphi^J$ that produce the same result but that do not share any task type between each other. Each application φ^j is composed of several tasks $\varphi^j_1, \ldots, \varphi^j_{I_j}$ such that a graph φ^j and a graph $\varphi^{j'}$ do not share any task type: $t(i,j) \neq t(i',j')$ for $1 \leqslant j, j' \leqslant J$ and $j \neq j'$ and for $1 \leqslant i \leqslant I_j$ and $1 \leqslant i' \leqslant I_{j'}$.

Obviously as this version of the problem includes the previous one, it is at least as complex, and thus this version is also unary NP-complete. In this section, we exhibit a dynamic program with pseudo-polynomial complexity to solve it, which proves that the problem is not binary NP-complete.

The throughput of task φ^j_i on a machine is denoted by $\rho_{j,i}$. To obtain the prescribed throughput, we may either use several application graphs concurrently or increase the throughput of a graph by renting several processors corresponding to its task types. Generally, an optimal solution is obtained as a combination of these two strategies, and the throughput of each graph φ^j has to be carefully tuned.

We now present a dynamic program with pseudo-polynomial complexity to compute an optimal solution in this case. It relies on $C(\rho, j)$, which denotes the minimum cost to reach a throughput ρ using only the first j application graphs (among the J defined in the model). $C(\rho, j)$ can be computed as follows:

$$C(\rho, j) = \begin{cases} \displaystyle\sum_{i=1}^{I_1} \left\lceil \frac{n^1_{t(i,1)}}{r_{t(i,1)}} \cdot \rho \right\rceil \times c_{t(1,k)} \text{ if } j = 1 \\ \displaystyle\min_{0 \leqslant \rho_j \leqslant \rho} \left(C(\rho - \rho_j, j-1) + \sum_{i=1}^{I_j} \left\lceil \frac{n^j_{t(i,j)}}{r_{(t(i,j))}} \cdot \rho_j \right\rceil \times c_{t(i,j)} \right) \\ \text{otherwise} \end{cases}$$

The base case $j = 1$ corresponds to a single application graph, and is thus similar to Section 4.1. In the general case, the prescribed throughput ρ is split into two parts: ρ_j, which is the throughput devoted to graph φ^j, and is computed as previously, and $\rho - \rho_j$, which devoted to the the first $j-1$ graphs, and thus can be expressed recursively. The final result $C(\rho)$ is obtained as $C(\rho, J)$. Note that as each processor of type q delivers an integer throughput r_q, the throughput of an application graph can only be an integer. Thus, there is a finite set of possible integer values ρ_j to test in the previous formulation. To compute a given $C(\rho, j)$, all $C(\rho', j')$ with $\rho' \leqslant \rho$ and $j' \leqslant j$ must be computed. Based on these values, the complexity of the elementary computation is $O(\rho I)$. The complexity of computing $C(\rho)$ is thus $O(\rho^2 IJ)$.

5.3 Applications with shared task types

Here, we consider a generalization of the previous setting, where tasks φ^j_k of different application graphs may have the same type:

$$\begin{aligned} &\exists\, j, j' (1 \leqslant j, j' \leqslant J, \neq j'), \exists\, i\ (1 \leqslant i \leqslant I_j), \\ &\quad \exists\, i' (1 \leqslant i' \leqslant I_{j'}) \text{ s.t. } t(i,j) = t(i',j') \end{aligned}$$

As a consequence, a processor may be shared between several application graphs. Although the implementation of this option is more difficult, due to the complex control needed, it is expected to give better performance since several (expensive) processors may be shared by all possible application graphs.

The whole throughput ρ must be at least the sum of the individual application graph throughput ρ_j:

$$\sum_{j=1}^{J} \rho_j \geqslant \rho \tag{1}$$

For each task type q we need to rent enough processors to reach the global throughput ρ.

$$\forall q \; x_q \cdot r_q \geqslant \sum_{j=1}^{J} \left(\sum_{i=1|t(i,j)=q}^{I_j} \rho_j \right), \qquad (2)$$

with x_q the number of instances of type q such that $q = t(i,j)$ and r_q throughput of processor P_q when executing a task of type q.

Then the problem can be formulated as the following MIP:

$$\begin{cases} \text{Minimizing } C(\rho) = \sum_{q=1}^{Q} x_q \cdot c_q \\ \text{under constraints (1) and (2)} \\ \text{with } x_q \in \mathbb{N} \end{cases}$$

Despite our efforts, we have not been able to determine if this general version is unary or binary NP-complete. However, given the additional complexity of splitting the prescribed throughput into a sum of elementary throughputs, we conjecture that this problem is binary NP-complete, contrarily to the previous version. In the following we propose several heuristics of polynomial complexity to address the problem.

6 Heuristics

In this section we propose several heuristics which address the problem presented in Section 5.3. Moreover, if possible, an optimal solution is computed using an integer linear program solver as described in the next section.

H0 (random) H0 randomly chooses each throughput ρ_j for each graph φ^j ($1 \leqslant j \leqslant J$) such that the constraint $\sum_{1 \leqslant j \leqslant J} \rho_j = \rho$ is satisfied.

H1 (best graph) The H1 algorithm selects only one application graph. It chooses the graph φ^j ($1 \leqslant j \leqslant J$) whose cost is minimum to reach the desired throughput, that is $\rho_j = \rho$. This cost is computed as in Section 4.1. The complexity of the H1 algorithm is in $O(J \times Q)$.

H2 (random walk) The H2 algorithm starts from the solution given by H1 $(\rho_1, \rho_2, \ldots, \rho_J)$ and tries to improve this solution iteratively. At each step, H2 randomly chooses two different graphs φ_{j_1} and φ_{j_2}, and then moves a fraction δ of the throughput from φ_{j_1} to φ_{j_2} such that their throughputs respectively become $\rho_{j_1} - \delta$ and $\rho_{j_2} + \delta$ for the next iteration. If $\rho_{j_1} < \delta$, ρ_{j_1} becomes equal to zero and ρ_{j_2} equal to $\rho_{j_2} + \rho_{j_1}$. This solution is stored if it improves the current minimal cost. In any case, this new solution is the starting point of the next iteration even if this computed solution does not reduce the current minimal cost yet. The heuristic stops after a predetermined number of iterations and outputs the best encountered solution.

H31 (stochastic descent) The H31 algorithm is very similar to the H2 heuristic. The main difference is that H31 retains the same solution for the next iteration as long as no improvement is obtained by the exchange. However, if the cost associated to the computed solution is lower than the current minimal cost, this new solution becomes the baseline solution for the next iteration. The heuristics stops if a given number of iterations is reached or if the solution corresponding to the minimal cost has not changed for a predetermined number of iterations.

H32/H32Jump (steepest gradient) The H32 and H32Jump algorithms follow the steepest gradient paradigm. As for the two previous heuristics, both the H32 and the H32Jump algorithms start using the solution given by H1. All possible throughput fraction exchanges between graphs are tested and only the one leading to the smallest platform cost is stored. When no improvement is possible, the current solution is a local minimum and is output by H32. To search for a better solution that this local minimum, H32Jump allows for a deterioration of the current solution by accepting a given number of throughput exchanges between graphs without checking if the solution is improved or not. Then, the improvement process described before is started again using the obtained current solution as the new baseline. This solution is in a neighborhood of the solution corresponding to last obtained local minimum. If this neighborhood is large enough, another local minimum may be found. This solution is stored if its cost is smaller than the previous stored minimum solution. The solution corresponding to the smallest cost from all the computed solutions is returned by H32Jump.

7 Illustrating example

In this section we illustrate on an example the ability of some heuristics to simultaneously use several application graphs in order to improve the platform cost. In this example we consider an application that can be described indifferently by three workflows as

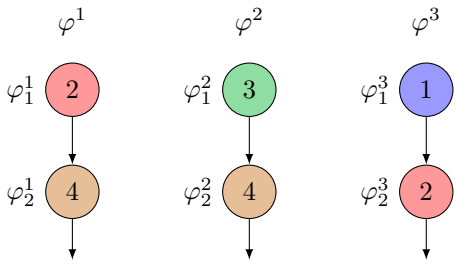

Figure 2: Illustrating example: Application parameters

Processor	type	ρ	cost
P_1	t_1	10	10
P_2	t_2	20	18
P_3	t_3	30	25
P_4	t_4	40	33

Table 2: Illustrating example: Available machines

shown in Figure 2. Whatever the graph that is used, one input leads the same output after its execution. Each workflow has two tasks and each task is of one type out of four. The available platform provides four machines, one for each task type, with different throughput performance and costs. The detailed parameters are shown in Table 2. Table 3 gathers the results of all heuristics. The first column contains the desired throughput ρ. The following columns present for the ILP and each heuristics the chosen throughputs (ρ_1, ρ_2, ρ_3) for each application graph and the corresponding solution cost. For example, for a desired throughput of $\rho = 70$, the solution chosen by the ILP splits the total throughput into $\rho_1 = 10$, $\rho_2 = 30$ and $\rho_3 = 30$, for the corresponding application graphs. For this solution, according to the platform parameters, one has to rent 3 instances of P_1, 2 instances of P_2, 1 instance of P_3 and 1 instance of P_4, with total cost of 124.

To highlight the optimal costs on Table 3, optimal values are printed in bold. It can easily be seen that in this small example, both heuristics H2 and H32jump very often find the optimal cost (H2 fails only twice). Moreover, it also shows the bucket behavior of H1: as H1 only chooses one application graph, the same solution may be chosen for one or more consecutive throughputs until no more idle capacity is available. In the case of a desired throughput $\rho = 160$ none of the heuristics is capable of finding the optimal cost. They all output the same solution with only one application graph, whereas the optimal solution uses all 3 available graphs.

8 Experiments

To assess the quality of the different heuristics we have developed a simulator in python. The heuristics described in Section 6 are implemented in this simulator and the integer linear programs are solved by calling the Gurobi[§] library from the simulator. In this section we present the simulator, the experimental settings and the obtained results.

8.1 The cloud renting simulator

The simulator aims at assessing the quality of the algorithms used to choose the best way to execute an application on the cloud when it is described indifferently by several graphs. These algorithms give the part of the target throughput each graph has to reach so as to decrease the cost of the whole platform as much as possible. Starting from a configuration file that gives the properties of the application graphs and the properties of the cloud, it randomly generates several sets of application graphs and their corresponding sets of machines. For each couple of application graph set and machine set, it then applies the heuristics and the ILP resolution to find a solution as cost effective as possible.

The relevant parameters when generating a set of application graphs and a cloud are dictated by the problem model. For the application graphs these parameters are:

- The number of tasks that compose an application graph: When generating an application graph its number of tasks is randomly chosen in the interval $[min\_tasks, max\_task]$. Thus all the application graphs do not get the same number of tasks. This is to avoid getting too similar applications.
- The number of available types that can be used: This number is a fixed value for the simulation. There is no need for an interval here as task types are randomly chosen.
- The number of application graphs: The number is a fixed value for the simulation.

In our first attempt, the application graphs for an application are randomly generated: For each task of an application graph its type is randomly chosen

[§] http://www.gurobi.com/

	I L P				H1				H 2				H 31				H 32				H 32 JUMP			
ρ	ρ_1	ρ_2	ρ_3	cost	ρ_1	ρ_2	ρ_3	cost	ρ_1	ρ_2	ρ_3	cost	ρ_1	ρ_2	ρ_3	cost	ρ_1	ρ_2	ρ_3	cost	ρ_1	ρ_2	ρ_3	cost
10	0	0	10	**28**	0	0	10	**28**	0	0	10	**28**	0	0	10	**28**	0	0	10	**28**	0	0	10	**28**
20	0	0	20	**38**	0	0	20	**38**	0	0	20	**38**	0	0	20	**38**	0	0	20	**38**	0	0	20	**38**
30	0	30	0	**58**	0	30	0	**58**	0	30	0	**58**	0	30	0	**58**	0	30	0	**58**	0	30	0	**58**
40	40	0	0	**69**	40	0	0	**69**	40	0	0	**69**	40	0	0	**69**	40	0	0	**69**	40	0	0	**69**
50	10	30	10	**86**	0	0	50	104	10	30	10	**86**	0	0	50	104	0	0	50	104	10	30	10	**86**
60	40	0	20	**107**	0	0	60	114	40	0	20	**107**	0	0	60	114	0	0	60	114	40	0	20	**107**
70	10	30	30	**124**	70	0	0	138	10	30	30	**124**	70	0	0	138	70	0	0	138	10	30	30	**124**
80	20	60	0	**134**	80	0	0	138	20	60	0	**134**	80	0	0	138	80	0	0	138	80	0	0	138
90	50	30	10	**155**	0	90	0	174	50	30	10	**155**	0	80	10	169	0	80	10	169	50	30	10	**155**
100	20	60	20	**172**	100	0	0	189	20	60	20	**172**	100	0	0	189	100	0	0	189	20	60	20	**172**
110	20	90	0	**192**	0	110	0	199	20	90	0	**192**	0	110	0	199	0	110	0	199	20	90	0	**192**
120	0	120	0	**199**	0	120	0	199	0	120	0	199	0	120	0	199	0	120	0	199	0	120	0	199
130	30	90	10	**220**	0	0	130	256	30	90	10	**220**	0	0	130	256	0	0	130	256	90	30	10	224
140	0	120	20	**237**	0	140	0	257	0	120	20	**237**	0	140	0	257	0	140	0	257	0	120	20	**237**
150	0	150	0	**257**	0	150	0	**257**	0	150	0	**257**	0	150	0	**257**	0	150	0	**257**	0	150	0	**257**
160	40	120	0	**268**	160	0	0	276	100	60	0	272	160	0	0	276	160	0	0	276	160	0	0	276
170	10	150	10	**285**	0	170	0	315	10	150	10	**285**	10	150	10	**285**	10	150	10	**285**	10	150	10	**285**
180	40	120	20	**306**	0	180	0	315	40	120	20	**306**	0	180	0	315	0	180	0	315	40	120	20	**306**
190	10	150	30	**323**	0	190	0	340	10	150	30	**323**	10	180	0	333	10	180	0	333	10	150	30	**323**
200	20	180	0	**333**	0	200	0	340	20	180	0	**333**	0	200	0	340	0	200	0	340	20	180	0	**333**

Table 3: Illustrating example: Results

in a set of types. The machine throughput and its price is also randomly chosen in an interval. Using this totaly random generation we do not control the efficiency of the machine and hence most of the application graphs are of no use because they use inefficient machines compared with the others. As a result we do not get a real competition between all graphs but rather between only very few of them, and often, only one single graph leads to the smallest cost for the whole throughput. With these configurations the H1 heuristic is usually able to find a very good solution, and the ILP quickly picks out the optimal solution, in particular when the requested throughput is high compared to the machine performance. To focus on the difficult and realistic cases, where some of the tasks of an application graph are replaced by other type of tasks (e.g. when a task running on GPU is replaced by a task running on a classical CPU architecture for a matrix product), we first randomly generate a initial application graph. The other, alternative, application graphs of the set are then generated by randomly changing a percentage of tasks of this initial graph. As a result the application graphs of the set that are able to perform the same outputs considering the same inputs share more tasks than totaly randomly chosen graphs.

The relevant parameters for the cloud generation are:

- The throughput of each machine: To generate different types of machines the throughput of each machine is randomly chosen in the interval $[min\_thrgpt, max\_thrgpt]$.
- The price of each machine: The price is chosen between 1 and a higher value.

For each parameter set, the cloud renting simulator generates hundred different configurations of applications and clouds. Then for each (application, cloud) configuration couple and for a set of target throughputs it computes the cost values obtained using the ILP and the heuristics.

8.2 Results

Using the cloud renting simulator we have tested several (application, cloud) configurations to assess the behavior of the ILP and the heuristics. There is no real interest to give all the results so we concentrate here on three of the most interesting parameter settings with small, medium and large application graphs. The tested target throughput ρ ranges from 20 to 200 with a step size of 10.

8.3 Small application graphs

With the first settings we consider the case of small application graphs. We have generated 20 alternative graphs per application that are able to perform the same outputs considering the same inputs. Each graph contains between 5 and 8 tasks. The percentage of tasks that are changed between the initial application graph and the alternative graphs is 50%. The cloud is composed of 5 different types of machines, each costing between 1 and 100 and delivering a throughput between 10 and 100.

Figure 3 shows the normalization of cost values obtained by the heuristics with the optimal solution computed by the ILP. The size of the addressed problem in this case is quite small and it is not surprising that the ILP is always able to compute the optimal solution even if the variables are integer. Note also that the results given by the heuristics are

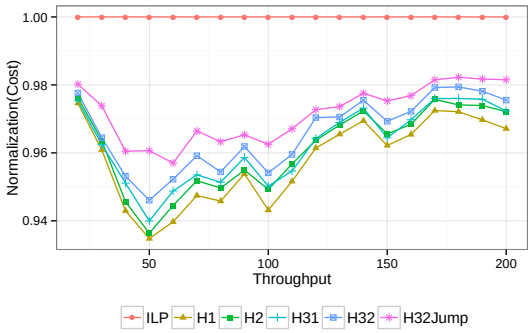

Figure 3: Normalization of cost with the optimal solution. (20 alternative graphs, between 5 and 8 tasks for each graph)

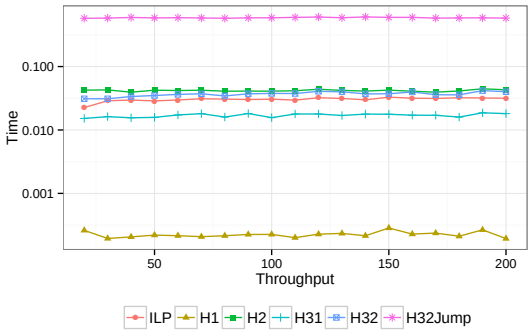

Figure 5: Computation time for the heuristics. (20 alternative graphs, between 5 and 8 tasks for each graph)

not far from the ILP solution, no more than 6%. The second element that we note is that a hierarchy exists between heuristics. The order between them remains the same throughout the entire experiment even if the baseline solution is given by H1. H32jump performs the best among our heuristics. Finally we take note that solutions given by H1 can always be improved.

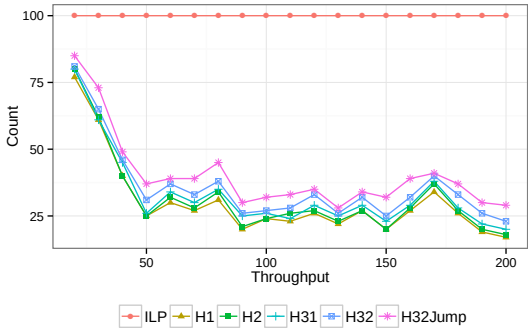

Figure 4: Number of times where each algorithm finds the best. (20 alternative graphs, between 5 and 8 tasks for each graph)

Figure 4 shows the number of times that each approach proposes the lowest cost value within the 100 simulations for each throughput value between 10 to 200. In this simulation the number of instance types and the size of application graphs are not large and the ILP still finds the optimal solution. However, Figure 4 shows that almost all heuristics also find the optimal solution in more than a quarter of the runs.

Figure 5 shows the run time of the heuristics and the ILP. H1 almost instantly finds its solutions. H31 is a little faster than the ILP. H2 and H32 follow very closely. H32jump is the slowest. As expected, H32jump is by far the slowest, but as already stated above, this heuristics achieves the best results apart from the ILP. This latter point can easily be explained by the fact that H32Jump performs several steepest gradient algorithms before giving its solution.

8.4 Medium application graphs

In the second setting we consider the case of medium application graphs. We have generated 20 alternative graphs per application and each graph has between 10 and 20 tasks. The percentage of tasks that are changed between the initial application graph and the alternative graphs is 30%. The cloud is composed of 8 different types of machines each costing between 1 and 100 and delivering a throughput between 10 and 100.

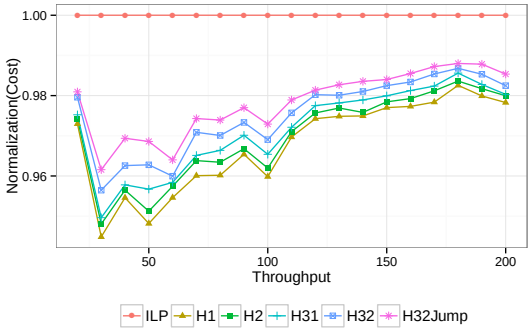

Figure 6: Normalization of cost with the optimal solution. (20 alternative graphs, between 10 and 20 tasks for each graph)

Figure 6 presents the normalized results of this second setting. As for small applications, the order within our heuristics stays the same: H1 – H2 – H31 – H32 – H32jump. The quality of the solutions given by the heuristics is also about the same as for small graphs, within 5% of the ILP solution.

8.5 Large application graphs

In the third setting we consider the case of medium application graphs. We have generated 20 alternative application graphs that contain between 50 and 100 tasks. The percentage of tasks that are changed between the initial application graph and the alternative graphs is 50%. The cloud is composed of 8 different types of machines each costing between 1 and 100 and delivering a throughput between 10 and 50.

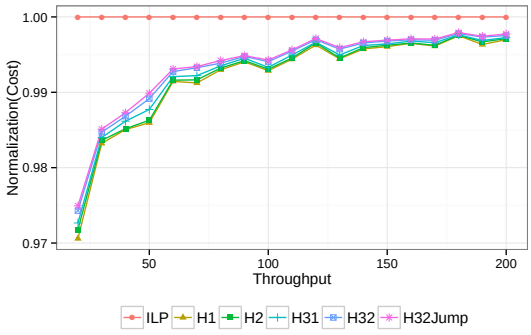

Figure 7: Normalization of cost with the optimal solution. (20 alternative graphs, between 50 and 100 tasks for each graph)

Figure 7 shows the normalization of cost values obtained by the heuristics with the optimal solution computed by the ILP. In this case the heuristics perform almost identically and the normalized cost shows a performance of more than 99% for throughputs higher than 50: for throughputs sufficiently large, using a single graph (such as the one output by H1) is enough to get close to optimal cost. The solutions given by H1 and H2/H32Jump becomes very close and if another graph is chosen, the higher the target throughput, the smaller its contribution. Indeed our heuristics become asymptotically close to the optimal cost value as the throughput is increasing considering such large application graphs.

In order to find the limits of the ILP we made a larger experiment where we limited the search time of the ILP to 100 s. In this experiment we have generated 10 alternative application graphs that are able to perform the same outputs considering the same inputs. Each graph contains between 100 and 200 tasks. The percentage of tasks that changed between the initial application graph and the alternative graphs is 30%. The cloud is composed of 50 different types of machines each costing between 1 and 100 and delivering a throughput between 5 and 25. Using this configuration the ILP is often not able to find the optimal results. In Figure 8 one can see that for a throughput larger than 100, the ILP reaches the time limit of 100 s. Note that increasing the time limit value from 100 s to 5 minutes (300 s) does not significantly improve the result. In this case the ILP still its current solution with smallest cost, but it is able to guarantee that it is optimal.

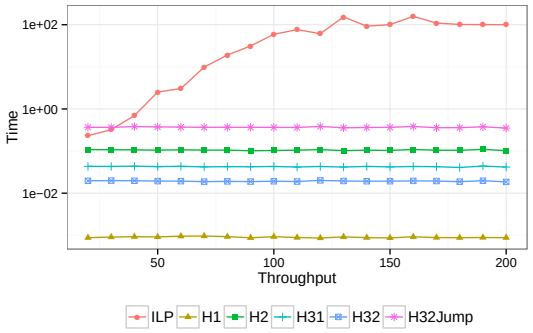

Figure 8: Computation time for the heuristics. (20 alternative graphs, between 100 and 200 tasks for each graph)

8.6 Summary

Our experiments show that even if the case of multi-DAG applications with shared task types is a NP-complete problem, an efficient ILP solver allows to compute optimal solutions, i.e., an allocation with the smallest cost, for small and medium sized solutions. On the other hand for applications with a large number of tasks, i.e., more than hundred tasks, the ILP fails to find the optimal solution. It generaly returns a good solution but not always the best. It also generates much longer computing times when the number of tasks increase.

In the lack of such a solver, the best graph heuristics given by heuristic H1 allows to compute allocations with minimal cost overhead (in most of the cases the overhead is less than 2% of the optimal solution). All in all improved heuristic solutions does not allow to achieve more than 5% over the naive H1 approach, in the tested configurations. We also

show that the naive H1 approach leads to solutions whose cost becomes asymptotically close to the optimal cost value when this one can be found using the ILP.

9 Conclusion

In this paper, we investigate the problem of renting resources on the cloud for one application that can be described in several ways, each one by one different application graph (DAG). Indeed, as resources on the cloud are massively heterogeneous, one application can be described by several graphs. The considered applications are workflow applications that have to achieve a target throughput. The issue is to find the suitable throughput distribution between DAGs and then the rented instances corresponding to these graphs corresponding to the part of the target throughput that they have to perform. We show that, in some cases, this problem can be optimally solved using dynamic programming approach even if the problem is known to be NP-complete in the weak sens (knapsack problem using negative loads). But this problem becomes unfortunately harder in the most general cases when application graphs can share tasks and thus machines that perform these shared tasks. The real complexity of the most general problem remains open (obviously at least NP-Complete in a weak sens). However we propose a characterization of the optimal solution by designing an Integer Linear Program. If an efficient solver is used, optimal solutions can be found as shown in the experimental section of the paper. As solving such an ILP can not be guaranteed, we propose several efficient heuristics that are able to reach solutions very close to the optimal solution in many cases as shown by numerous simulations. Our approaches lead to solutions that are less than 6% from the optimal value and this percentage decreases when the target throughput increases. For future work we plan to test our approach on real applications and we will then try to integrate our solution as a pre-step before the deployment phase in existing Cloud deployment systems like Pegasus or CometCloud.

Acknowledgement

This work has been supported by the Labex ACTION project (contract "ANR-11-LABX-01-01") and the ANR DATAZERO (contract "ANR-15-CE25-0012"). All computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[2] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *6th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2004*. IEEE Computer Society Press, 2004.

[3] A. Benoit, U. V. Çatalyürek, Y. Robert, and E. Saule. A Survey of Pipelined Workflow Scheduling: Models and Algorithms. Technical Report RR-LIP-2010-28, LIP, ENS Lyon, France, Sept. 2010.

[4] A. Benoit, A. Dobrila, J.-M. Nicod, and L. Philippe. Scheduling linear chain streaming applications on heterogeneous systems with failures. *Future Generation Computer Systems*, 29(5):1140 – 1151, 2013.

[5] A. Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. In *Cluster Computing, 2007 IEEE International Conference on*, pages 497 –506, sept. 2007.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[7] J. J. Durillo, R. Prodan, and W. Huang. Workflow Scheduling in Amazon EC2. In *Euro-Par Workshops'13*, pages 374–383, 2013.

[8] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-Degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10. IEEE, Nov. 2008.

[9] S. Genaud and J. Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011*, pages 1–8, 2011.

[10] F. Guirado, A. Ripoll, C. Roig, A. Hernàndez, and E. Luque. Exploiting throughput for pipeline execution in streaming image processing applications. In *Proceedings of the 12th international conference on Parallel Processing*, Euro-Par'06, pages 1095–1105, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] T. D. R. Hartley, A. Fasih, C. A. Berdanier, F. Özgüner, and Ü. V. Çatalyürek. Investigating the use of GPU-accelerated nodes for SAR image formation. In *CLUSTER*, pages 1–8, 2009.

[12] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

[13] S. Pandey, L. Wu, S. M. Guru, and R. Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010*, pages 400–407. IEEE Computer Society, 2010.

[14] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[15] G. Singh, C. Kesselman, and E. Deelman. Optimizing grid-based workflow execution. *Journal of Grid Computing*, 3:201–219, 2005.

[16] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[17] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: pricing in the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[18] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang. A market-oriented hierarchical scheduling strategy in cloud workflow systems. *The Journal of Supercomputing*, 63:256–293, 2013.

[19] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic resource allocation for spot markets in clouds. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.