Cost-Aware Scheduling of Deadline-Constrained Task Workflows in Public Cloud Environments

Hendrik Moens*, Koen Handekyn† and Filip De Turck*
* Ghent University – iMinds, Department of Information Technology
Gaston Crommenlaan 8/201, B-9050 Gent, Belgium
† UP-nxt, Vaartstraat 14/1, 3000 Leuven, Belgium
e-mail: hendrik.moens@intec.ugent.be

***Abstract*—Public cloud computing infrastructure offers resources on-demand, and makes it possible to develop applications that elastically scale when demand changes. This capacity can be used to schedule highly parallellizable task workflows, where individual tasks consist of many small steps. By dynamically scaling the number of virtual machines used, based on varying resource requirements of different steps, lower costs can be achieved, and workflows that would previously have been infeasible can be executed. In this paper, we describe how task workflows consisting of large numbers of distributable steps can be provisioned on public cloud infrastructure in a cost-efficient way, taking into account workflow deadlines. We formally define the problem, and describe an ILP-based algorithm and two heuristic algorithms to solve it. We simulate how the three algorithms perform when scheduling these task workflows on public cloud infrastructure, using the various instance types of the Amazon EC2 cloud, and we evaluate the achieved cost and execution speed of the three algorithms using two different task workflows based on a document processing application.**

***Keywords*—Public clouds, Scheduling, Workflow deployment**

I. Introduction

The rise of cloud computing enables elastic on-demand resource provisioning. This technology makes it possible to improve existing applications, or to create new types of applications that were previously not feasible. A class of services that can be improved using cloud technologies are document processing applications. Document processing applications handle large amounts of documents, and apply different tasks, such as extraction of metadata, processing, logging and archiving, to the documents. While individual documents only require a small number of different steps, and a limited amount of computation time, very large batches are processed at once, leading to significant system requirements. The processing of these batches is subject to performance constraints in the form of deadlines.

While these workloads could be executed using a fixed set of servers over a long time period, the elasticity of cloud environments allows its users to vary the amounts and types of servers during execution. This elastic scaling of resources in public clouds allows the application to better exploit the potential workload parallelization, thus executing tasks faster, making it possible to achieve deadlines that would previously not have been possible. The pay-as-you-go nature of public clouds on the other hand ensures only the resources that are

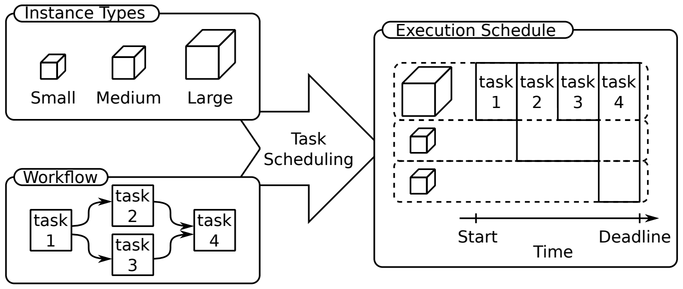


Fig. 1: The Public Cloud Cost-Aware Workflow Scheduling (PCCAWS) problem takes as input a description of the public cloud instance types and a workflow description, and returns the cheapest schedule that can be used to plan the tasks, making use of the different cloud instance types.

actually used are paid for, potentially reducing the cost of executing the workflow.

This public cloud task scheduling differs from regular scheduling techniques, as different billing modes are used, and the number of resources that can potentially be used is virtually unconstrained. Furthermore, the presence of multi-core instances increases the problem complexity: as many of the offered servers contain multiple cores, it is possible to dedicate individual tasks to specific cores, thus running multiple tasks in parallel on a single server. The billing models used within our problem formulation are based on those of Amazon EC2, Microsoft Windows Azure and Rackspace, where different instance types can be used, and where the cost of using instances is billed for every interval (typically an hour) during which the server is used. We only focus on cost-aware scheduling of tasks by users of public cloud infrastructure, billing itself is handled by existing mechanisms implemented by the cloud provider.

In this paper, we focus on the scheduling of task workflows on public cloud infrastructure, a problem we refer to as the Public Cloud Cost-Aware Workflow Scheduling (PCCAWS) problem. We determine how a single workflow, consisting of multiple tasks such as unzipping data files, validating, processing and archiving, can be provisioned in a cost-efficient way. The different tasks themselves can be split into many parallelizable steps. In other words, we focus on scheduling *workflows* which consist of several *tasks*, and where each task consists of several individual *steps*. These steps can be executed independently from each other. As the steps of the task can be executed independently, the task itself is parallellizable. We take information concerning the workflow, such as task complexity, relations and deadlines, and information related to the public cloud infrastructure such as the available instance types, and their performance and cost into account. An overview is shown in Figure 1.

Within this paper we address three research questions: (i) How can the PCCAWS problem be represented formally? (ii) Which heuristic and optimal approaches can be formulated to solve the PCCAWS problem? and (iii) What is the performance, both in achieved cost and execution speed, of these heuristic algorithms compared to an optimal ILP-based algorithm? To achieve this, we developed a formal mathematical model, which we implemented using a commercial Integer Linear Programming (ILP) solver, and two heuristic approaches. The result of the optimization algorithm is a schedule that contains the different instance types used by the solution, which describes when the different task steps are executed on each instance. To evaluate this approach we used it to plan the load of multiple large workflows in a document processing use case. The resulting schedule can be used for two purposes: (1) it can be used purely to schedule the task workflow, by creating different Virtual Machines (VMs) based on the schedule and running them, and (2) it can be used to estimate the cost with which it is 100% certain that the task workflow can be executed.

In the next Section, we will discuss cloud and scheduling related work. Afterwards, in Section III we will describe the formal problem model. Section IV describes both optimal and heuristic algorithms. Subsequently, Section V describes the evaluation setup, which is followed by the evaluation in Section VI. Finally, Section VII describes our conclusions.

II. Related work

Our approach has similarities to different cloud task scheduling approaches [1], [2], [3], but focuses on the scheduling a large number of small tasks, which requires a more finegrained control than the usual VMs used in these approaches. In effect, our algorithms determine the tasks that must be grouped together in one VM.

The problem we discuss differs from surge computing [4], [1], where parts of applications and workflows spill over from a private cloud to a public cloud, and from partial cloud migration [5], because in our approach, the entire workload is short-lived and executed in its entirety on the public cloud environment. The short-lived nature of individual steps ensures they do not need to be migrated during their execution. Our approach could be adapted to be used in hybrid cloud environments consisting of private and public environments, or in an environment containing multiple clouds, but then network costs would have to be added to the model.

In cloud environments, application placement [6], [7] is used to determine on which physical machine application instances are executed. In our approach, the granularity of the tasks that must be placed is much higher, while the tasks themselves are shortlived. Furthermore, only the instance type must be determined, and which tasks are executed by which instance. After this, the underlying cloud environment will make use of application techniques to actually place the resulting instances.

In [1], the authors focus on cost-optimal resource provisioning of batch workloads with deadlines, and define a binary integer problem to solve the provisioning problem. While the authors take time aspects into account, this is in 1 hour timeslots. We however focus on task workflows consisting of a very large number of small parallellizable tasks, for which much more finegrained time information is needed. We also present heuristic approaches to solving the problem that scale better than ILP-based solutions.

Our work has similarities to scheduling problems for production processes. Many such scheduling problems are presented as online problems, but this restriction is not needed as the tasks are known before execution. Furthermore, the presented problem differs from classical on-line scheduling problems [8], [9], as in our approach, the potential amount of resources is infinite, different resource types with varying performance are possible, and that goal is cost minimization rather than makespan minimization, as the latter would be trivial when infinite resources are available. Our approach further differs form generic production scheduling because the cost formulation of using servers in clouds is particularly complex, as we use server cores as resources for task execution, while it is server use that impacts billing intervals.

There are also similarities to project scheduling problems, where a schedule is determined to execute a sequence of tasks by a given deadline on a specified amount of resources. The main difference with the resource constrained project scheduling problem [10], is again that the amount of usable resources in a cloud environment is virtually unconstrained, and that computing the cost of the used resources is non-trivial. Furthermore, determining the duration of tasks is also more complex, as task duration is influenced by the machine upon which the task is executed. The problem we consider further differs from these problems as the number of steps executed in a workflow is much larger.

III. Formal model

The various symbols used in this section are shown in Table I. We will first describe the optimization objective, after which the different model variables are discussed. Finally we will describe the different constraints used in the model.

A. Optimization objective

The objective of the model is to determine a configuration where all tasks are executed at a minimal cost. For every server instance i, this cost can be determined by calculating the product of the cost of using a server during an interval, $\mathrm{intervalCost}(i)$, with the number of intervals during which the server is used, referred to as $\mathrm{intervals}(i)$. The final expression is shown in Equation (1), where these costs are summed for all servers instances $i \in I$.

$$\min\left(\sum_{i\in I}\mathrm{intervalCost}(i)\times\mathrm{intervals}(i)\right) \tag{1}$$

B. Model parameters

The model makes use of a collection of server types S, that represent the different types of servers offered by the cloud platform. Each server type $s \in S$ has a billing interval $\text{interval}(s)$ and a cost $\text{intervalCost}(s)$, which is billed for every instance of the server type for every interval during which the instance is used. For most cloud providers, this billing interval equals one hour for all available server types.

A solution contains a collection of servers instances I. Each instance $i \in I$ has a type $s \in S$, referred to as $\text{type}(i)$. While the number of servers within the cloud is assumed to be unlimited, the model makes use of a finite set of servers. We make use of a simple algorithm to determine a feasible small server collection, which we will discuss in Section III-D. It is possible for some of these servers to remain unused.

A collection of cores C, on which tasks are scheduled is used. Each instance i of type $s \in S$ has a set of one or more cores, referred to as $\text{cores}(i)$. The number of cores a server has is determined by its server type, and is represented by $\text{coreCount}(s)$. Every core of a server of type $s \in S$ has a capacity, referred to as $\text{capacity}(s)$, that determines the amount of operations a core can execute per time unit.

All of the properties of servers, their cores, and the number of cores, are contained in the server type, of which the server is an instance. In the following equations, these properties will need to be used for both servers and cores. To shorten these equations, we make it possible to refer to properties of server types using the core c or instance i as arguments. When a property, such as capacity is applied using an instance $i \in I$ as an argument, the server type is determined using $\text{type}(i)$. Similarly, when a core $c \in C$ is used as an argument, the instance $i \in I$ for which $c \in cores(i)$ is used.

Tasks consist of multiple steps that can be executed in parallel. A task $t \in T$ contains $\text{stepCount}(t)$ steps, and each individual step has a computational complexity $\text{complexity}(t)$, which corresponds to the required CPU cycles of step. Different tasks can be dependent on each other: if $\text{before}(t_1, t_2)$, the task t_2 can only start after all of the steps of task t_1 have finished executing.

C. Model constraints

Each task has a start time, and an end time, indicating when the task starts and ends its execution. Each task also has start and end times on each of the cores c. The task start time occurs before the task starts executing on any of the cores, and similarly, tasks end after their execution on all cores have ended. Finally, every task must end before the deadline, D. This is expressed in Equations (2) to (5), which are added for all tasks $t \in T$ and cores $c \in C$.

$$\text{start}(t) \leq \text{start}(t, c) + D \times \text{coreUnused}(t, c) \tag{2}$$
$$\text{start}(t, c) \leq \text{end}(t, c) \tag{3}$$
$$\text{end}(t, c) \leq \text{end}(t) + D \times \text{coreUnused}(t, c) \tag{4}$$
$$\text{end}(t) \leq D \tag{5}$$

These three constraints are only relevant for cores that are actually used. Because of this, the equations make use of an additional variable, $\text{coreUnused}(t, c)$, which takes on value 1

TABLE I: The different symbols used in Section III.

Input Variables		
Symbol	Type	Description
S		The set of server types.
I		The set of server instances.
C		The set of server cores on which individual execution threads can run.
T		The set of tasks.
$\text{type}(i)$	S	The type of an instance $i \in I$.
$\text{coreCount}(s)$	$\mathbb{I}_{>0}$	The number cores in a server of type s.
$\text{cores}(i)$	$\mathcal{P}(C)$	A set containing the cores of an instance i.
$\text{capacity}(s)$	$\mathbb{R}_{>0}$	The amount of operations a core of server type s can execute per time unit.
$\text{interval}(i)$	$\mathbb{R}_{>0}$	The time interval by which the use of an instance i is measured.
$\text{intervalCost}(s)$	$\mathbb{R}_{>0}$	The cost of using a server of type s during an interval.
$\text{stepCount}(t)$	$\mathbb{I}$	The number of undividable steps in task t.
$\text{complexity}(t)$	$\mathbb{R}_{>0}$	The processing capacity required for executing a step of a task t.
D	$\mathbb{R}_{>0}$	The deadline by which the tasks must be completed.
Decision Variables		
Symbol	Type	Description
$\text{start}(t)$	$\mathbb{R}_{\geq 0}$	The time when the execution of task t starts.
$\text{end}(t)$	$\mathbb{R}_{\geq 0}$	The time when the execution of task t ends.
$\text{start}(t, c)$	$\mathbb{R}_{\geq 0}$	The time when the execution of task t starts on core c.
$\text{end}(t, c)$	$\mathbb{R}_{\geq 0}$	The time when the execution of task t ends on core c.
$\text{startSU}(i)$	$\mathbb{R}_{\geq 0}$	The moment in time when the instance i starts being used.
$\text{endSU}(i)$	$\mathbb{R}_{\geq 0}$	The moment in time when the instance i stops being used.
$\text{processed}(t, c)$	$\mathbb{I}$	The number of steps of a task t that are processed on a core c.
$\text{intervals}(i)$	$\mathbb{I}$	The number of intervals of billing time during which the instance i is used.
$\text{ord}(t_1, t_2, c)$	$\mathbb{B}$	This binary variable has value 1 if task t_1 is executed before task t_2 on core c, and 0 otherwise.
$\text{coreUnused}(t, c)$	$\mathbb{B}$	This binary variable has value 1 is task t does not make use of core c, and value 0 otherwise.

if a core is not used, and 0 otherwise. If $\text{coreUnused}(t, c)$ takes on value 1, the task deadline D is added to the inequalities in Equations (2) and (4), which ensures they are always true. To ensure the coreUnused variables have correct values, we make use of Equation (6). This equation ensures coreUnused can only take on value 1 if the start and end times are equal.

$$\text{end}(t, c) - \text{start}(t, c) \leq D \times (1 - \text{coreUnused}(t, c)) \tag{6}$$

As previously mentioned, relations between tasks exist. If a task t_1 must end before another task t_2 can start, expressed as $\text{before}(t_1, t_2)$, the relation $\text{end}(t_1) \leq \text{start}(t_2)$ is added to the model.

The model is used to allocate tasks on cores at a specific time. The following equations are used to ensure all tasks are correctly allocated. The duration of a task $t \in T$ on a server of type $s \in S$ can be determined as shown in Equation (7).

$$\text{duration}(t, s) = \frac{\text{complexity}(t)}{\text{capacity}(s)} \tag{7}$$

This constant is then used in Equation (8) which links the task start and stop times to the number of steps of task t that are executed on core c. Finally, Equation (9) ensures that the number of steps that are processed is equal to the step count of the task, thus ensuring all steps are allocated on a core.

$$\text{end}(t,c) - \text{start}(t,c) = \text{processed}(t,c) \times \text{duration}(t,c) \tag{8}$$

$$\sum_{c \in C} \text{processed}(t,c) = \text{stepCount}(t) \tag{9}$$

To determine the cost of using a server instance, it is important to know when it is used for the first and last times. The first use of a server is before any core starts being used, expressed in Equation (10), while the server stops being used after the last core has been used, which is expressed in Equation (11).

$$\text{startSU}(i) \leq \text{start}(t,c) \tag{10}$$

$$\text{end}(t,c) \leq \text{endSU}(i) \tag{11}$$

The cost of using servers is measured in intervals. To determine the number of intervals used, we add Equation (12) for every instance $i \in I$.

$$\text{intervals}(i) \times \text{interval}(i) \geq \text{endSU}(i) - \text{startSU}(i) \tag{12}$$

To support parallel execution of tasks, two additional constraints are needed to prevent tasks from occurring concurrently. For this we introduce additional variables that determine the order of tasks on a core. If $\text{ord}(t_1, t2, c)$ is 1, t_1 is executed before t_2 on core c, if its value is 0, the tasks occur in the opposite order. This is expressed in Equations (13) and (14). These additional variables and constraints are only added if it is possible for two tasks to be executed concurrently.

$$\text{end}(t_2,c) - \text{ord}(t_1,t_2,c) \times D \leq \text{start}(t_1,c) \tag{13}$$

$$\text{ord}(t_1,t_2,c) = 1 - \text{ord}(t_2,t_1,c) \tag{14}$$

D. Initial server configuration

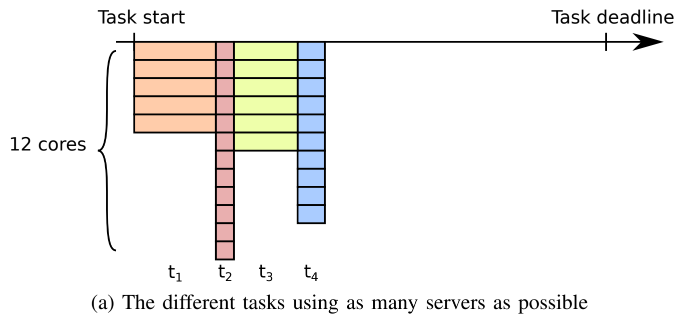
The model makes use of a limited set of server instances I, based on a set of server types S. The set I must be chosen in such a way that there are enough servers to lead to a feasible flow, but not too many servers ensuring the algorithm execution does not become too complex. For every server type $s \in S$ that exists, we make a feasible configuration as though only servers of that type are used. This can be used to determine the minimal number of servers of the type that are required.

Initially, we place all tasks sequentially, ensuring they maximally use parallelization, executing only one step for every core. This ensures each task t makes use of $|C|_t = \text{stepCount}(t)$ cores. From this number of cores, we can then determine the number of servers $|I|_t = \lceil \frac{|C|_t}{\text{coreCount}(s)} \rceil$ of type s are needed. Using $\max_{t \in T} |I|_t$ would yield a possible bound for the number of servers of type s which could be used within the model. However, as in the document processing use case some tasks may contain in the order of 10^5 or more steps, a tighter bound is needed.

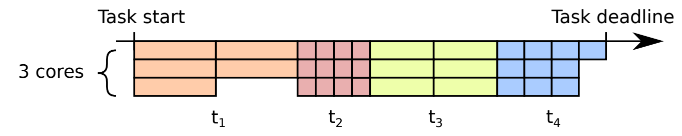
We use the Server Count Bound (SCB) algorithm, shown in Algorithm 1, to obtain a smaller estimate on the number of CPU cores. The first phase of the algorithm determines a maximum number of cores, as discussed in the previous paragraph, illustrated in Figure 2a. Afterwards, the tasks using

```
Data: server type $s \in S$
for $t \in T$ do
    $|C|_t \leftarrow \text{stepCount}(t)$;
end
$\sigma \leftarrow \sum_{t \in T} duration(t, s)$;
while true do
    $|C| \leftarrow \max_{t \in T} |C|_t$;
    if $|C| = 1$ then return 1;
    $T^{high} \leftarrow t \in T : |C|_t = |C|$;
    $\delta \leftarrow \sum_{t \in T^{high}} \text{duration}(t, |C| - 1) - \text{duration}(t, |C|)$;
    if $\sigma + \delta <= D$ then
        for $t \in T^{high}$ do
            $|C|_t \leftarrow |C| - 1$;
        end
        $\sigma \leftarrow \sigma + \delta$;
    else
        return $|C|$;
    end
end
```

Algorithm 1: The SCB algorithm: an approach for determining a tighter bound on the number of servers.



(a) The different tasks using as many servers as possible



(b) The result of the packing algorithm

Fig. 2: An illustration on how an upper bound to the number of servers can be determined. A block represents a single step within a task.

the most servers are iteratively rescheduled, lowering the number of cores needed, but increasing the execution time. This process is repeated, until further rescheduling of tasks would make the task workflow exceed their deadline. The result of this process for the example is illustrated in Figure 2b.

The algorithm maintains a collection of values $|C|_t$ that, for every task t, contains the number of cores on which it is scheduled. Initially, $|C|_t$ equals the number of steps in the tasks. The algorithm also maintains the current task execution duration σ throughout its execution. In each iteration, the algorithm determines the tasks that currently use most cores. It then determines the impact, on the time needed for executing the tasks, δ, of decreasing the number of cores used by 1. If $\sigma + \delta$ remains less than D, the core count is decreased and the next iteration is started. Otherwise, the algorithm terminates and the maximum number of cores used is returned. The algorithm makes use of the duration of a task $t \in T$ when it is executed on n cores, which we define in Equation (15).

$$\text{duration}(t, n) = \left\lceil \frac{\text{stepCount}(t)}{n} \right\rceil \times \text{duration}(t, c) \quad (15)$$

IV. Algorithms

A. Integer Linear Programming: PCCAWS$_{ILP}$

The model discussed in Section III was used to construct an optimal algorithm, which we refer to as PCCAWS$_{ILP}$, using the CPLEX[11] ILP solver. ILP models make use of infinite-precision values that can not be represented correctly using computers. In practice, ILP solvers such as CPLEX make use of approximations, allowing slight violations of model constraints. These effects can however lead to infeasible results, where new tasks are started before the tasks on which it depends being finished, in particular for more difficult problems.

CPLEX allows users to define a non-zero parameter ϵ, and guarantees constraint violations will always be less than this ϵ. If, for example, a constraint $a \leq b$ exists within the model, it is possible that $a > b$ in the solution, but it is guaranteed that $a \leq b + \epsilon$.

By adding ϵ to the model constraints, we ensure correct results are achieved. Specifically, we extend Equation (7), and increase the duration of an individual task by 2ϵ, resulting in a modified equation shown in Equation (16).

$$\text{duration}'(t, c) = \frac{\text{complexity}(t)}{\text{capacity}(c)} + 2\epsilon \quad (16)$$

B. Server count bounds filling heuristic: PCCAWS$_{SCB}$

The SCB algorithm, can easily be extended into a simple heuristic. For every type of server that exists in the system, a schedule is created, making use of only this type of server. In this schedule, every task execution duration is chosen as long as determined by the SCB algorithm discussed in Section III-D. Effectively, the time allocated for a task by the algorithm is filled with executions of the task steps. The final result of the basic heuristic is determined by choosing the cheapest out of the different schedules.

The algorithm execution speed can be improved by using multithreading: the largest part of the algorithm execution consists of generating different schedules for every server type, a task that can be parallelized. We refer to this heuristic algorithm as PCCAWS$_{SCB}$.

C. Result improving heuristic: PCCAWS$^{+}_{SCB}$

The PCCAWS$_{SCB}$ algorithm is capable of finding a simple, feasible schedule. The result can however be improved by automatically applying small transformations to the intermediate schedules generated by the basic heuristic. The PCCAWS$^{+}_{SCB}$ algorithm automatically tries to execute two improvements to the result of the basic algorithms:

- *Clustering reassignment:* All servers of the same server type are split into their individual cores, after which the resulting cores are clustered based on their start and end use times. For this clustering, we make use of a modified k-means clustering, which is adapted to limit the cluster size. The resulting clusters of cores are then combined into new servers. This approach ensures the use of cores within a server have similar start and end times, which prevents cores from being underused within a server.
- *Server type changes:* This algorithm analyses every server in the schedule, and tries to replace it with a server of a cheaper server type. When cores are unused, or switching to a lighter, weaker server is possible, this approach ensures the cost of the schedule is reduced. This approach is particularly useful in cases where larger servers with more cores are cheaper to use than equally powerful servers containing less cores.

The PCCAWS$^{+}_{SCB}$ heuristic tries both improvements separately and in sequence, and subsequently returns the schedule with the lowest resulting cost value.

V. Evaluation setup

In the evaluations, we make use of the Amazon EC2 instances as described in [12]. The performance of the different instances is expressed by making use of the *EC2 Compute Unit*, a reference unit used by Amazon to compare CPU performance of different instances based on the CPU capacity of a 2007 Opteron or 2007 Xeon processor [12]. Using this information, we can estimate the execution times of a task on the different instance types based on the execution times of the task on a small instance (`Amazon_m1.s`). The costs of using these different instances is chosen based on information available from Amazon [13]. An overview of the considered instance types and prices is shown in Table II. We exclude Amazon micro instances, as they do not have reliable computational power, making them less suited for intensive workloads. We refer to this set of server types as *Amazon*.

It is of note that within a specific server type, costs are always multiplied by two for doubling the number of cores of a server. This implies that using an `Amazon_m1.xl` instance is equivalent to using two `Amazon_m1.l` instances. In this

TABLE II: The Amazon instance types, referred to as the *Amazon* server type set. (Capacity $\times$ Amazon EC2 Compute Unit, Cost per hours in \$/100)

Type	Property	Value
Standard instances	Cores	{1,1,2,4}
`Amazon_m1.{s, m, l, xl}`	Capacity	{1,2,2,2}
	Cost	{8, 16, 32, 64}
High memory instances	Cores	{2,4,8}
`Amazon_m2.{xl, 2xl, 4xl}`	Capacity	{3.25, 3.25, 3.25}
	Cost	{45, 90, 180}
High CPU instances	Cores	{2, 8}
`Amazon_c1.{m, xl}`	Capacity	{2.5, 2.5}
	Cost	{16.5, 66}
Cluster compute instances	Cores	{8, 16}
`Amazon_cc1.{4xl, 8xl}`	Capacity	{8, 16}
	Cost	{130, 240}

TABLE III: The evaluation workflows. (Step durations as executed on an `Amazon_m1.s` instance)

Name	Task	Steps	Step Duration (s)
Sequential_n	Unzip	1	35.0
	Extract	$\lfloor 2 \times 10^5/n \rfloor$	$0.5 \times n$
	Process	$\lfloor 2 \times 10^5/n \rfloor$	$0.2 \times n$
	Zip	1	90.0
Deadline	120 min		
Relations	Unzip $\rightarrow$ Extract $\rightarrow$ Process $\rightarrow$ Zip		
Parallel_n	Unzip	1	35.0
	Extract	$\lfloor 2 \times 10^5/n \rfloor$	$0.5 \times n$
	Process	$\lfloor 2 \times 10^5/n \rfloor$	$0.2 \times n$
	Zip	1	90.0
	Log	$\lfloor 2 \times 10^5/n \rfloor$	$0.1 \times n$
Deadline	120 min		
Relations	Unzip $\rightarrow$ Extract $\rightarrow$ Process $\rightarrow$ Zip		
	Unzip $\rightarrow$ Log $\rightarrow$ Zip		

case it makes no sense to consider anything but the smallest instances, as if in an optimal solution a larger instance is used, an equivalent solution can be determined using multiple smaller instances, and the more instances, the more time the algorithm needs to execute. We call the set containing only the smallest instance types from within each group $Amazon_{small}$. Within the evaluations, the lowest cost achieved by $Amazon_{small}$ will be the same as that of $Amazon$, but as less server types are considered, the result can be determined faster.

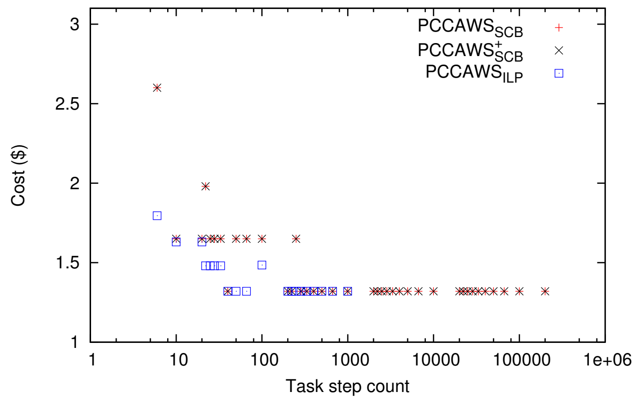
In practice, it would generally be preferable to use the larger instances when possible, as managing large numbers of servers introduces an overhead. To model this, we define an additional set $Amazon_{mc}$, which is based on the $Amazon$ set, but where a small management cost of \$0.01 is added for all servers. This ensures that larger servers are chosen when possible.

We evaluate the presented algorithms using the Stevin Supercomputer Infrastructure at Ghent University. Each node on which the algorithms were executed contains dual-socket 2.5 GHz quad core Intel Xeon L5420 processors, thus having 8 cores, and 16 GB memory. To prevent the multi-tenant nature of the cluster from impacting execution speed measurements of the heuristic algorithms, they are measured using 100 executions on an Ubuntu server with Intel Core i3 2.93GHz processor and 4GiB memory.

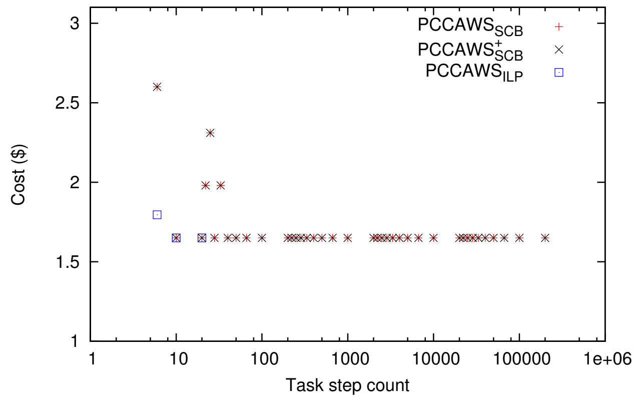
The evaluation workflows are shown in Table III. They are characterized by their name, and an integer value n. The latter is used to create multiple versions of the workflow, each having a similar execution load and deadline. Increasing the n value decreases the number of tasks, and increases individual task durations. This can be seen as grouping multiple tasks together, e.g. always executing 1000 steps at once. This decreases the granularity of tasks, potentially resulting in more expensive configurations, but also decreases the complexity of the algorithms. The steps, step duration and step counts are estimates of the workload of a basic document processing flow. Note that the parallel workflow contains almost 50% more tasks than the sequential workflow for the same n value.

VI. Evaluation Results

We evaluate the costs and execution speed achieved by the PCCAWS$_{SCB}$, PCCAWS$^{+}_{SCB}$ and PCCAWS$_{ILP}$ algorithms for the Sequential_n and Parallel_n problems. Varying the n value of both problems influences the duration of steps and the number of steps in the workflows. A large n value ensures many steps are grouped together, decreasing the number of steps considered by the algorithms. A smaller n value increases the number of steps used in the model, making it possible to execute a more finegrained planning. In this section, we will determine the cost and execution speed in function of the resulting task step count for a subtask (with a maximum of 2×10^5 steps when $n = 1$). All presented graphs were generated for n values from 30000 to 1. As we found that the results vary more for lower n values, the datapoints were chosen accordingly, ensuring there are more datapoints for lower n values. The range of points is chosen so that the task step count remains between 2×10^5, the maximum number of steps of the workflow, and 6, a minimal number of steps. Smaller step counts were not considered, as those would increase step execution times too much, making it impossible to create a feasible schedule that still respects the deadline.



(a) Sequential workflow



(b) Parallel workflow

Fig. 3: The cost of executing the workflow on the public cloud using the $Amazon$ and $Amazon_{small}$ server type sets. Task step count $= \lfloor 2 \times 10^5/n \rfloor$

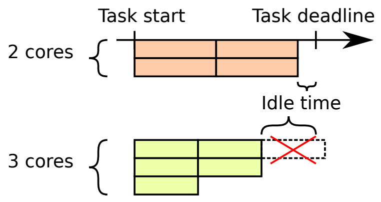


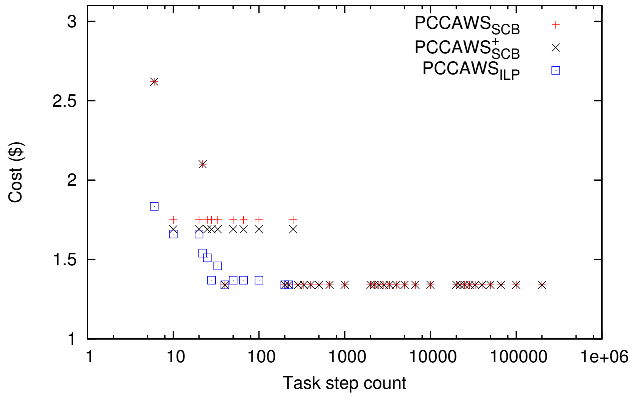
Fig. 4: Increasing the number of steps while decreasing their execution time can, sometimes, increase the number of cores needed in a schedule.

A. *Algorithm costs*

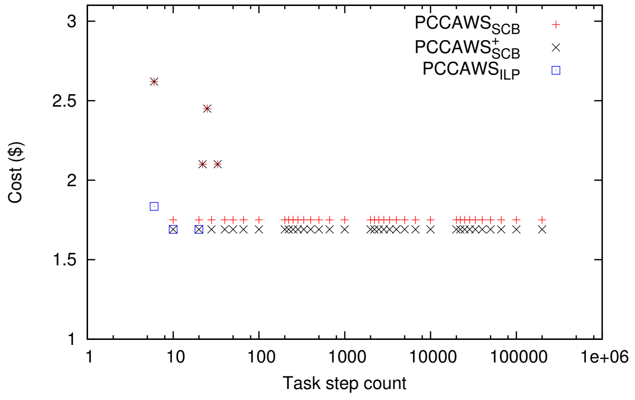
The cost achieved by the different algorithms for the $Amazon_{small}$ server type set is shown in Figure 3. The quality of the larger set $Amazon$ was measured as well, and, as expected, yields exactly the same quality results. Figures 3a and 3b show the sequential and parallel problems respectively. The PCCAWS$_{ILP}$ algorithm is shown only for lower task step counts, as increasing the step count causes the memory and time needed for execution to increase significantly. As can be seen in the figures, for the $Amazon$ and $Amazon_{small}$ server type sets, the PCCAWS$_{SCB}$ and PCCAWS$^{+}_{SCB}$ algorithm both lead to the same cost, which is higher than the cost achieved by the PCCAWS$_{ILP}$ algorithm for small numbers of task step counts. As the step count increases, the cost achieved by all three of the algorithms decreases until it levels out at the same cost. This decrease is not monotonous, as sometimes an increased step count causes worse solutions to be chosen. These increases in cost also occur at times for the PCCAWS$_{ILP}$ algorithm, indicating this is inherent to the problem itself, and not just a property of the heuristic algorithms. The cause is illustrated in Figure 4, and can be explained as follows: if, for a chosen step size, all steps can be executed on a small collection of cores, slightly decreasing the step size, and thus increasing the number of steps, can in some cases only increase the idle time, and not the number of steps executed per core. As there are however more steps that must be executed, this can cause the need to add extra cores, which in turn can increase the cost of the solution.

When we consider the $Amazon_{mc}$ server type set, where more differences occur between different server types, the PCCAWS$_{SCB}$ and PCCAWS$^{+}_{SCB}$ algorithms at times yield different results, as shown in Figures 5a and 5b for the sequential and parallel problems respectively. In the case of the sequential problem, both algorithms still achieve the same end result, but for the parallel workflow, the PCCAWS$^{+}_{SCB}$ algorithm achieves better results. Again we find that, for low task step counts, the PCCAWS$_{ILP}$ algorithm tends to outperform the other algorithms, but as step counts increase, the difference between both decreases.

In all of the considered cases, the PCCAWS$^{+}_{SCB}$ algorithm, when executed for the full workflow containing 2×10^5 tasks, obtains the same cost as the best value that could still be computed by the PCCAWS$_{ILP}$ algorithm. When the PCCAWS$_{ILP}$ algorithm is used, good cost estimates can be made, even when many tasks are grouped together.



(a) Sequential workflow



(b) Parallel workflow

Fig. 5: The cost of executing the workflow on the public cloud using the $Amazon_{mc}$ server type set.

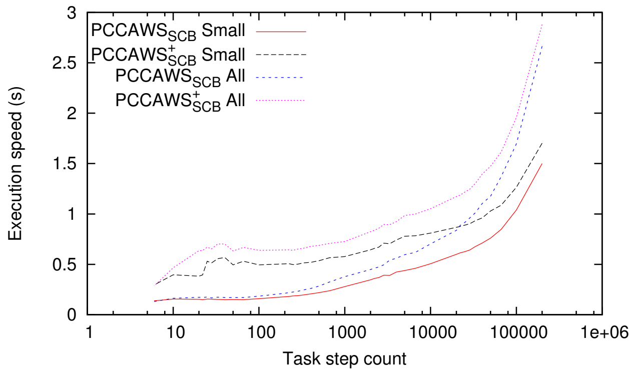


Fig. 6: The execution times of the algorithms for the $Amazon$ an $Amazon_{small}$ server type sets for the parallel workflow.

TABLE IV: Execution speed of the PCCAWS$_{ILP}$ algorithm. Times are in seconds.

	Sequential_n		Parallel_n	
	Avg	σ	Avg	σ
$Amazon$	4854.4	8864.9	4407.0	6219.0
$Amazon_{small}$	259.6	810.4	10991.0	15485.4

B. Algorithm execution speed

In Figure 6, we compare the execution times of the PCCAWS$_{SCB}$ and PCCAWS$^{+}_{SCB}$ algorithms applied to the parallel workflow, for both the full set of servers $Amazon$ and the small set of servers $Amazon_{small}$. The figure shows that that, for all algorithms, as the number of steps increases, so does the time needed for the algorithms to execute. Despite this, even for the maximum task step count of 2×10^5, the execution times for the different heuristic algorithms remain between 1.5 to 3 seconds. We also see that the PCCAWS$_{SCB}$ algorithms execute slightly faster than the PCCAWS$^{+}_{SCB}$ algorithms, and that using the limited server set $Amazon_{small}$ requires less resources than the larger set $Amazon$. When instead the sequential workflow is used, an identical trend is observed, but the flows generally execute slightly faster, taking between 1 to 2 seconds to execute for the maximum task step count. These execution times for the sequential workflow are omitted due to space constraints, and as they are nearly identical to Figure 6. The results indicate that PCCAWS$_{SCB}$ and PCCAWS$^{+}_{SCB}$ both scale linearly in the amount of task steps.

For completeness we also give an indication of the execution times of the PCCAWS$_{ILP}$ algorithm in Table IV. Note that unlike the previous execution speed measurements these times were evaluated using the Stevin Supercomputer Infrastructure, as the server used for the previous evaluations did not have the required memory and computational capacity to solve the ILP problem. Because of the long execution time of the PCCAWS$_{ILP}$ algorithm, it was executed only once for every problem, which explains the high standard deviation. We observe that the PCCAWS$_{ILP}$ algorithm requires a prohibitively long time to execute, even when tasks require only small numbers of steps when large n values are chosen, and that for smaller n values, the problem can not be solved due to insufficient memory. This limitation is particularly noticeable for the Parallel_n workflow, as there n values less than 10^4 failed to execute.

While the PCCAWS$^{+}_{SCB}$ algorithm only slightly improves upon the PCCAWS$_{SCB}$ algorithm, it requires only limited additional execution time, making it a useful addition to the basic algorithm. Especially as, for both considered workflows, the PCCAWS$^{+}_{SCB}$ algorithm is, for higher task step counts, capable of determining the same result as the best result that could still be computed by the PCCAWS$_{ILP}$ algorithm.

VII. Conclusions

In this paper, we formally described a model for scheduling workflows consisting of multiple parallellizable tasks on public cloud infrastructure, a problem we referred to as the PCCAWS problem, and described an optimal and two heuristic algorithms to solve the problem. We evaluated the algorithms through simulations using workflows based on those used in a document processing applications and the Amazon instance types. We found that, when individual steps within the workflows require a relatively long time to execute, the optimal algorithms outperform the heuristic algorithms with regards to achieved cost. As individual tasks become smaller however, the best heuristic algorithm was capable of finding a solution with the same cost as the best cost which could be determined by the optimal algorithm. We also found that the heuristic algorithms scale well, as all evaluations were executed in at most 3 seconds.

Acknowledgement

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is partly funded by the iMinds CUSTOMSS[14] project, and was carried out using the Stevin Supercomputer Infrastructure at Ghent University, funded by Ghent University, the Hercules Foundation and the Flemish Government, department EWI.

References

[1] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads," in *2010 IEEE 3rd International Conference on Cloud Computing*. Ieee, Jul. 2010, pp. 228–235.

[2] J. Tordsson *et al.*, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Generation Computer Systems*, vol. 28, no. 2, pp. 358–367, Feb. 2012.

[3] J. L. Lucas-Simarro *et al.*, "Scheduling strategies for optimal service deployment across multiple clouds," *Future Generation Computer Systems*, pp. 1–11, Jan. 2012.

[4] M. Armbrust *et al.*, "Above the Clouds : A Berkeley View of Cloud Computing," University of California at Berkley, Tech. Rep., 2009.

[5] M. Hajjat *et al.*, "Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications tothe Cloud," in *Proceedings of the ACM SIGCOMM 2010 conference*, 2010, pp. 243–254.

[6] J. Rolia, A. Andrzejak, and M. Arlitt, "Automating enterprise application placement in resource utilities," in *Self-Managing Distributed Systems: 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2003*. Springer, 2004, pp. 118–129.

[7] F. Wuhib, R. Stadler, and M. Spreitzer, "Gossip-based Resource Management for Cloud Environments," in *6th International Conference on Network and Service Management (CNSM)*, 2010, pp. 1–8.

[8] R. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, 1966.

[9] S. Albers, "Better bounds for online scheduling," *Proceedings of the twenty-ninth annual ACM*, vol. 29, no. 2, pp. 459–473, 1997.

[10] A. A. Lazarev and E. R. Gafarov, "On project scheduling problem," *Automation and Remote Control*, vol. 69, no. 12, pp. 2070–2087, 2009.

[11] (2012) IBM ILOG CPLEX 12.4. [Online]. Available: http://www-01.ibm.com/software/integration/optimization/cplex-optimizer

[12] (2012) Amazon EC2 instance types. [Online]. Available: http://aws.amazon.com/ec2/instance-types/

[13] (2012) Amazon EC2 pricing. [Online]. Available: http://aws.amazon.com/ec2/pricing/

[14] (2012) CUSTOMSS: CUSTOMization of Software Services in the cloud. [Online]. Available: http://www.iminds.be/en/projects/overview-projects/p/detail/customss