Improving the simulation of IaaS Clouds

Luke Bertot

Université de Strasbourg

École doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire ICube

THÈSE présenté par :

Luke Bertot

Soutenue le : **17 Juin 2019**

pour obtenir le grade de : **Docteur de l'université de Strasbourg**
Discipline / Spécialité : Informatique

Improving the simulation of IaaS Clouds.

Amélioration de simulation de cloud IaaS via l'emploi de méthodes stochastiques.

THÈSE dirigée par :
Stéphane GENAUD, professeur des universités, université de Strasbourg

Co-Encadrant :
Julien GOSSA, maitre de conférences, université de Strasbourg

---

Rapporteurs :
Laurent PHILIPPE, professeur des universités, université de Franche-Comté
Christophe CÉRIN, professeur des universités, université Paris XIII

Autres membres du jury :
Adrien LÈBRE, professeur, Institut Mines-Télécom Atlantique

Acknowledgments

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

Contents

Introduction

Motivations

The recent evolution in virtualisation technologies in the early 2000s has brought about new models for the exploitation of computing resources. These new models, often regrouped under the umbrella of cloud computing, along with the ever increasing ubiquity of internet access have transformed the hosting and development landscape over the last fifteen years.

Infrastructure as a Service (IaaS) cloud has revolutionized the *hosting* landscape, by making it possible to provision resources of different sizes on the fly. For hosting providers virtualisation-based provisioning allowed for more homogeneous data-centers and shorter turnaround times on rentals. For users IaaS makes it possible to keep a smaller baseline infrastructure, only raising capacity as necessary, while only paying their actual platform usage.

This new economic model has been hugely successful in the context of service providers, where services are meant to run perpetually and scaling is done to adjust to demand. In the context of finite workloads such as those seen in scientific computing using IaaS clouds poses additional challenges. One such challenge is the need for specialized setups, like in the cases of workloads designed to run on HPC clusters and workloads designed for grids and batch schedulers that are more easily executed on the cloud environments. However such workloads often require external scheduling. Such a scheduler needs to be adapted to the *on-demand* and the *pay-as-you-go* nature of IaaS clouds.

On institutional grids and clusters users are usually sheltered from the cost of computing resources. Access policies can range from being granted a fixed quota of resources to completely open access. The hidden cost falls on the institution in the form of its investment in infrastructure and manpower. The *pay-as-you-go* nature of the cloud means that to budget their experiment the user must be able to predict the quantity of computing resources needed. Moreover if they failed to do so the cloud would allow them to continue using resources while charging them potentially over the expected budget. This is only made harder by the *on-demand* nature of the cloud which allows for different scheduling approaches, some of which have non-trivial impact on workload execution time or cost.

Enabling scientists to use the IaaS cloud to execute scientific workloads will require:

A Cloud Scheduler capable of handling the workloads and managing the cloud re-

sources for the users. Such a tool must be able to schedule the workload to the cloud as well as provisioning and releasing cloud resources. Doing so automatically diminishes the risk of mismanagement resulting in higher costs.

A Prediction Tool capable of offering an estimate of the cost of executing a workload. This tool must be able to take into account the different available scheduling strategies to give the user the opportunity to choose the strategy meeting their needs.

Contribution

This work deals with the problem of building reliable prediction tools. We aim to build such a tool using IaaS cloud simulators. As with other distributed systems, a number of cloud simulation toolkits have been developed. Our team has long worked to evaluate the effectiveness of such simulators as prediction tools. We performed comparisons of real executions in different clouds and their corresponding simulations to find which parameters have the biggest impact on simulation precision.

This approach was successful in creating simulations that very closely reproduce their corresponding runs. However the use of such simulations as a prediction tool is still impeded by the inherent variability in the execution of applications on distributed systems such as Infrastructure as a Service (IaaS) clouds. Executing the same workload multiple times on the cloud will not always give the same execution length or cost. Accounting for this variability is a difficult problem, that most simulators do not address. This lapse not only reduces the user's confidence in the simulation, but reduces the quality of the information offered to the user.

Our thesis is that using stochastic simulation with a single parameter we can account for the variability observed in the executions of scientific applications on the IaaS cloud. In this work we propose a statistical approach to task scheduling on IaaS clouds. Our simulator accounts for the variability of the underlying cloud through a single parameter and presents the user with distributions of possible outcomes with a high level of confidence. This method can easily be used to extend other simulators. We use this method to predictively compare scheduling strategies when executing workloads on IaaS clouds, to help the user choose the strategy that best satisfies their time and cost constraints. We show that our simulator is capable of capturing 90% of real executions.

Outline

In Part I of this work we discuss the recent evolutions of IaaS cloud computing and the different types of scientific workloads. In Section 1.3 we present a couple of scientific workloads we used in our experiments. Chapter 2 presents Schlouder, a cloud broker designed to execute batch jobs on IaaS clouds. In Chapter 3 we present SimSchlouder a simulator built to help Schlouder users find which scheduling heuristic matches their needs. We study SimSchlouder's accuracy in Section 3.3.

In Part II we propose using stochastic simulations as an improvement on our previous deterministic simulation. Chapter 4 presents our attempt to build one such simulation using the Monte Carlo methods. In Section 4.3 we propose one such simulation, a corresponding input model and we evaluate it through an empirical experiment. In Chapter 5 we study more precisely the effect of the Monte Carlo simulation (MCS) variables on the simulation's precision. And in Chapter 6 we study a case in which our MCS failed to produce satisfactory results and discuss the limits of this approach.

Part I

Background

Chapter 1

Context

1.1 Cloud Computing

In computing *virtualisation* refers to methods and technologies designed to deviate the execution environment from the real environment in which the execution actually happens. These techniques oppose the *virtual* environment thus created to the *physical* existing resources. In this work we concern ourselves with cases where virtualisation is used to split a large physical resource set in smaller virtual ones.

Although virtualisation has been historically associated with high computational overhead, over the last decades the integration of hardware-assisted virtualisation in CPUs has greatly reduced the impact of virtualisation and led to the emergence of new economic and exploitation approaches of computer resources in the form of Cloud computing. In the Cloud computing model, operators make a large pool of resources available on demand to users, usually in a *pay-as-you-go* fashion. Most Cloud offers can be classified in three broad categories. The Software as a Service (SaaS) offers provide a web-application, or a light front-end to a server side application, to their users. The Platform as a Service (PaaS) cloud is usually built around providing automatic deployment and hosting to user code. Last, Infrastructure as a Service (IaaS), which this works focuses on, provides physical resources such as computing power or storage space to the users. IaaS providers operate large datacenters, containing large servers and storage. The large Physical Machines (PMs) are split between the users by way of Virtual Machines (VMs). VMs constrain the available resources to a subset of the resources present on the PM on which the VM is hosted. A large choice of VM configurations, called flavors, allows the users to build a cluster of the appropriate size for their needs. Within a VM the users have full administrative access. Resources are paid by pre-established intervals, called billing time unit (BTU), that range from the minute to the month, with most operators billing hourly, and any started BTU being billed at full price.

Historically the service provided by IaaS clouds was the purview of *hosting providers*. Much like IaaS cloud providers, hosting providers operate large datacenters and rent part of their resources piecemeal to users. Both of them allow users to set-up their infrastructure without having to invest in the setting up of a datacenter or a server room, nor in

the additional personnel necessary to maintain and operate such infrastructures. High uptime installations are expensive in infrastructure, with redundant power and cooling systems, and in specialised personnel. This is especially problematic for users whose primary activity does not involve much computer infrastructure. Hosting providers could mutualise such costs creating affordable access to high-availability server-grade hardware. The main difference between the historical hosting solutions and IaaS cloud resides in the granularity of the rental.

Virtualisation has allowed IaaS providers to become more effective hosting providers. Since hosting services rented PMs, they had to possess every machine they rented out. Because of this, hosting providers had to support a lot of heterogeneous hardware to satisfy every users' needs. Misestimated demands could lead to providers running out of a certain machine type while another type of machine remained unused taking up rack space. Setup times for new rentals could take up to a few days depending on machine availability and rentals were operated on a monthly basis. Since VMs are an arbitrary subset of PM resources, IaaS operators do not need to operate heterogeneous hardware to satisfy user needs. On the contrary IaaS operators are encouraged to run large PMs capable of handling any type VMs the operator provides. This setup maximises VM availability and PM usage. Moreover, the time needed to set up a new VM is much shorter than a PM, meaning as long as the datacenter is not at full capacity providing a new VM is almost instantaneous. This is also what enables the finer grained rental periods (BTUs).

The ability to provision resources on the fly has changed the hosting landscape. Users are now able to face unexpected computational load by changing their infrastructure instantly. Either by upgrading to a more powerful VM, referred to as vertical scaling, or by adding more VMs to the infrastructure, called horizontal scaling. Depending on one's uses of the cloud, scaling can be used in different fashions. For some users, this can be to try to achieve a high level of service availability, like with a website. Scaling is used in this case to face fluctuation in demand. Whereas previous infrastructure had to be sized to handle the highest expected traffic load, users can now size their infrastructure on baseline traffic only bringing in additional resources during usage spikes. This greatly reduces the cost of keeping the service online for the user. For users using resources for computational purposes, scaling allows for a fine control of parallelization. This can be used to achieve shorter makespans (i.e. the time between the submission of the first task and the completion of the last one) at equal cost.

The advantages in infrastructure efficiency for the providers and usage flexibility for users led to the quick expansion of IaaS clouds. Today most of the historical hosting providers also provide virtualised hosting.

1.2 Scientific Computing

Computers' ability to deal with large amounts of data and computation has opened a lot of new possibilities in all fields of scientific research. Since most of these computations need not be sequential, parallelization is often used to improve and speed up these scien-

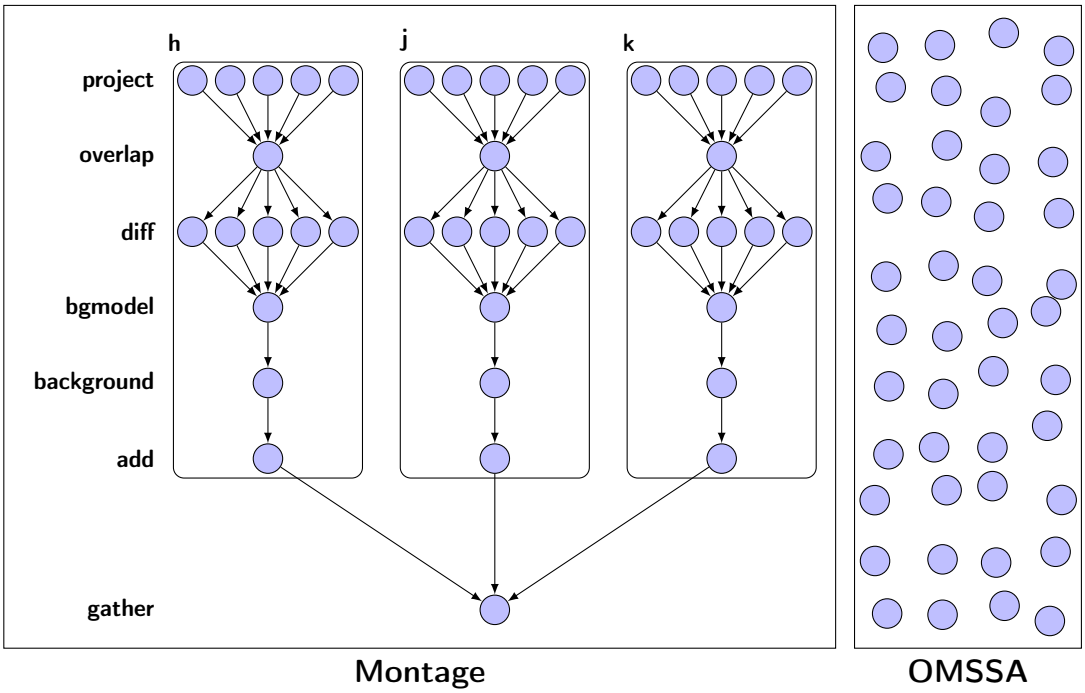

Figure 1.1: Two types of batch jobs. On the left a workflow (Montage [22]). On the right a bag-of-tasks (OMSSA [21]).

tific workloads. Parallelization can be performed in different fashions depending on the granularity of parallelization and the degree of control wanted.

At one end of the spectrum we find workloads designed for High Performance Computing (HPC). Composed of a single executable or a set of closely linked executables, an HPC application divides the workload into a set of computing tasks. These tasks are assigned to all available resources from the beginning to the end of the run. The tasks, that run in parallel and cooperate (for instance through message passing) are generally instances of a same program parameterized by the resource id the program runs on. HPC is usually done on dedicated hardware, and using programming constructs or languages adapted for parallel computing. Notable constructs and language used for HPC include MPI, often used conjointly with OpenMP and GPGPUs where relevant, or PGAS languages (such as X10 and Chapel). Such programs are often written with specific hardware in mind. Because data is exchanged between different processes at runtime, HPC programs must be executed on every node of the cluster synchronously. Although provisioning hardware similar to an HPC cluster is possible in some IaaS clouds, performance remains noticeably inferior to what can be obtained on a dedicated platform ([37, 42]).

On the other end of the spectrum we find batch jobs. These jobs are composed of sequential tasks designed to be executed independently. We classify these workloads in two categories depicted figure 1.1 :

- **bag-of-task** workloads have no dependencies between their tasks. The tasks can be executed in any order and with any level of parallelism.
- **workflows** exhibit data dependencies between tasks. This means the output of a task

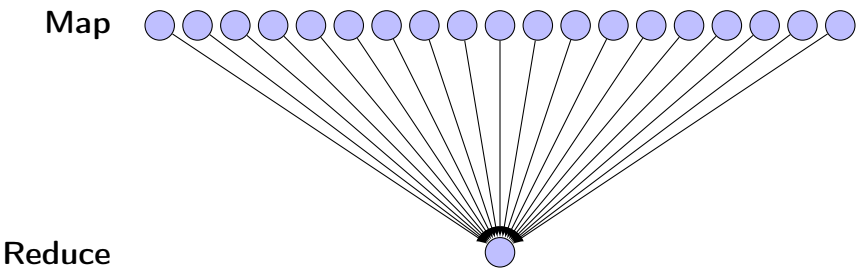

Figure 1.2: A typical Map Reduce workflow.

can be the input of another task. This constrains the execution order of tasks, however batch jobs do not require tasks be executed simultaneously. Workflows can be represented as directed acyclic graphs (DAGs) as in figure 1.1.

It is possible to find advanced workflow structures such as cases where some tasks are only executed when their dependencies results meet certain criteria. However such workflows are beyond the scope of this work. We will limit ourselves to cases where the dependencies and tasks do not change during the workflow's execution.

Batch jobs are designed to be executed on a large array of equipment, therefore code specialisation is limited to multicore usage when applicable. Since batch jobs are some set of separate programs, they need to be externally scheduled to be executed on the available machines. Outside of the data-dependencies in workflow the scheduler is afforded a lot of leeway in whether to parallelise the execution of a batch job or not. These scheduling decisions also depend on the available hardware. Historically batch jobs have been computed on resources belonging to universities or research institutions. Either using idle workstations during the off-hours or by investing into dedicated clusters to treat the jobs for the whole institution. Since the early 2000's, to face the ever increasing computing power needed, institutions have started mutualizing resources into shared infrastructures termed *grids*, accessed through specialized schedulers. HPC clusters however are rarely used to execute batch jobs since running non-specialised code is seen a wasteful usage of this very expensive equipment. The use of the external scheduler coupled with tasks capable of running on a wide array of hardware setups makes batch jobs a prime candidate for using IaaS cloud.

HPC workloads and batch jobs are the two ends of a spectrum, and some solutions exist in-between. One notable example of such a middle ground solution is found in Hadoop Map-Reduce jobs. Hadoop Map-Reduce is specialised in a single type of of workflow usually operating on a large input. As shown figure 1.2 a typical Map-Reduce job consists in :

- a large number of Map tasks, independent from one another and each operating on a different part of the input.
- a single Reduce tasks taking the Map's output as input.

Some variations exist, such as using multiple Reduce tasks when dealing with extremely large datasets, or repeating the whole Map-Reduce process for iterative applica-

tions, for instance frequently found in graph algorithms like Page Rank or shortest path computations. Map-Reduce jobs share characteristics with HPC, such as using a single binary running different functions on the various nodes. It also shares some of the batch jobs characteristics, such as adaptability to heterogeneous hardware. Although map tasks are completely independent, the Reduce tasks tries to collect results of finished maps on the fly to reduce the transmission overhead between the last map task and the reduce task. As with batch jobs, this kind of workload is prime for use of IaaS clouds. In fact some IaaS operators such as Amazon Web Service already propose the rental of preconfigured Hadoop clusters.

1.3 Sample applications

In the context of our work we conducted experiments that required running scientific workloads. To do so we used two applications from different scientific fields with wildly different execution profiles.

1.3.1 OMSSA

The first application we use comes from the field of proteomics. It is a data interpretation software called *Open Mass-Spectrometry Search Algorithm* (OMSSA) [21]. The proteomists and computer scientists of the Hubert Curien Pluridisciplinary Institute in Strasbourg have interfaced OMSSA with a production pipeline called *Mass Spectrometry Data Analysis* (MSDA), accessed through a web portal [12]. For its users, such a tool automatizes the parallel execution of several OMSSA instances distributed over the EGI grid in the Biomed virtual organization. This results in a higher analysis throughput and allows for large scale high throughput proteomics projects. Our team later ported this grid-based solution to an IaaS-cloud environment.

The tandem mass spectrometry analysis (also known as MS/MS analysis) consists in the fragmentation of peptides generated by enzymatic digestion of complex mixtures of proteins. Thousands of peptide fragmentation spectra are acquired and further interpreted to identify the proteins present in the biological sample. Peak lists of all measured peptide masses and their corresponding fragment ions are submitted to database search algorithms such as OMSSA for their identification. Each spectrum is then searched against a database of proteins. As each search is independent from the others, a natural parallelization consists in making this application a bag-of-task (BoT). The MSDA web portal wraps the different tasks in a workflow that manages data as well as parallel computations. It first copies the appropriate databases to a storage server, then groups the spectra into files to be sent to different computation sites. For each spectra file, a job running OMSSA is created to perform the identification search. All jobs can be run independently on different CPUs. The parallelization grain (i.e., the number of spectra distributed to each CPU) per OMSSA execution is computed by MSDA as a function of the requested resolution and the number of available CPUs.

To experiment with the execution of OMSSA, proteomists shared with our team four

Use-case	Tasks	Spectra (per task)	Input (MB) mean	Input (MB) sd	Output (MB) mean	Output (MB) sd	Runtime (s)(measured) mean	Runtime (s)(measured) sd
BRS	223	1250	2.6 (1.2)	0.4 (0.8)	3.4	1.6	603.1 (355.9)	54.9 (182.1)
BRT	33	10000	16.6	2.4	12.4	1.7	125.9	16.6
HRS	65	1250	1.3 (0.4)	0.2 (0.2)	8.9	5.9	182.3 (104.3)	6.8 (19.3)
HRT	34	10000	1.7	0.6	4.1	1.4	9.6	2.3

Table 1.1: Key characteristics of the OMSSA tasks in the different use-cases. Tasks with a lower number of spectra than the indicated one have their input sizes and runtime presented in parentheses.

datasets to use as input of OMSSA. These datasets are used to produce 4 OMSSA use-cases based on input data and OMSSA configuration. As described in the previous paragraph each set is composed of a number spectrometry results, each of which are, for parallelization purposes, split into files containing a given number of spectra. The main OMSSA use-case used in our work is called the BRS. BRS runs the most complete peptide search possible on a low resolution set composed of 33 base spectrometer results. These results are split using a granularity of 1250 spectra per file as recommended by the proteomists. This split results in 6 to 8 files per spectrometer result leading to a total workload of 223 tasks. The input size of tasks processing a full 1250 spectra averages 2.6MB with a standard deviation of 0.4MB, variations in file size comes from the format used to store the results. For the files containing less than 1250 spectra the average size 1.2MB with a 0.8MB standard deviations. The output file size appears not to be correlated to input size. The average output file size is 3.4MB and the standard deviation 1.6M. The execution of the workload on our cloud platform gives us a feel for the distribution of runtimes, with data transfers excluded. The average runtime for tasks possesing the full 1250 of spectra is 603.1s with a standard deviation of 54.9s. File with lower number of spectra average 355.9s with a standard deviation of 182.1s. Taking the workload globally, without any parallelization the complete execution of the 233 tasks would require an average of 35 hours and 10 minutes, of which the communications needed to download input files and upload output files represent less than 1%.

Table 1.1 present all the use-cases available to our team. Execution statistics were collected during executions done on our local platform. The first columns up to Output are constant from one execution to the other, whereas runtime will vary depending on the platform and execution. For BRS and HRS which both use lower numbers of spectra part tasks we measured input size and runtimes separately for tasks that don't have a full number of spectra to analyse. HRT and BRT have such a high number of spectra per tasks that data are not split into multiple tasks.

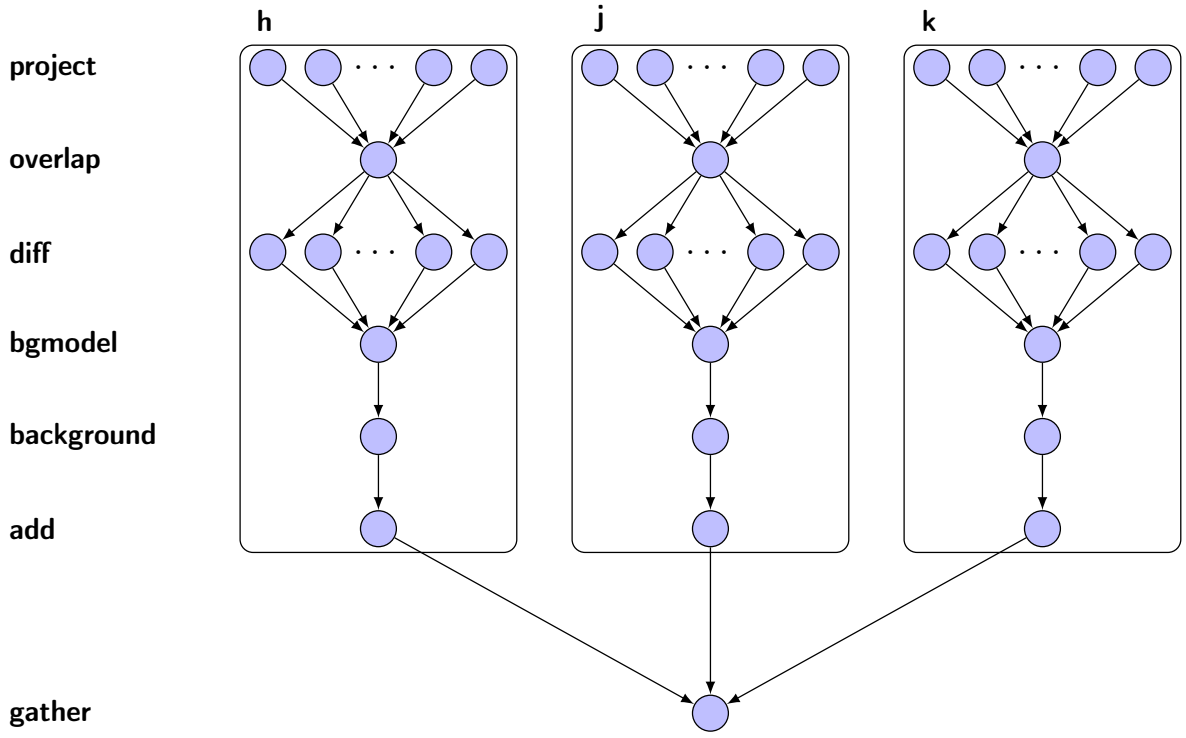

Figure 1.3: Directed Acyclic Graph of the Montage workflow

1.3.2 Montage

The second application comes from the field of astronomy. Montage [22] was developed at the California Institute of Technology (CalTech) with funding from the NASA and the National Science Foundation. Montage was designed from the ground up to be executed on a wide range of infrastructures including grids. The California Institute of Technology operates a web-portal allowing registered users to operate the workflow on a CalTech operated grid.

The composition of image mosaics is essential to astronomers. Most astronomical instruments have a limited field of view and a limited spectral range. This means that large astronomical structures can not be observed in a single shot. Composing a final image of a large astronomical structure will often require fusing images taken at different times, from different places, with different instruments. Although temporality and position are not huge factors in terms of the observed subject, astronomical structures do not usually change overnight and Earth's displacement is mostly negligible, aligning images from such different places and times requires special tools. The Montage Image Mosaic Toolkit (Montage) is one such tool. To generate such mosaics Montage performs two main calculations, re-projection and background rectification. Re-projection computes for each pixel of each input image pixels of the output image, while compensating for shape and orientation of the inputs. Background rectification aims to remove discrepancies in brightness and background, this helps compensating for differences in observation devices, time of night and ground position. Montage is a workflow of specialized tasks presenting strong data dependencies.

Task	Count	Input Size (MB)		Output Size (MB)		Runtime (s)	
	(per run)	mean	sd	mean	sd	mean	sd
project	108	14.6	0.8	68.8	10.0	9.1	1.7
overlap	3	0.1	0	0.2	0.00	2.5	0.7
diff	63	423.2	59.9	0.01	0.001	7.2	5.0
bgmodel	3	0.3	0.00	0.00	0	9.8	0.8
background	3	2476.1	9.9	2380.1	9.9	10.6	10.6
add	3	2476.1	9.9	1870.6	0.7	115.5	47.5
gather	1	4096	0	4.4	0	133.4	6.7

0: Zero, these inputs and outputs are always the exact same size.

0.00: Non-zero values inferior to the kB.

Table 1.2: Key characteristics of the Montage tasks in our usecase.

Our dataset for this workflow comes from the 2 Micron All-Sky Survey (2MASS) performed by the University of Massachusetts between 1997 and 2001. This survey was performed on 3 different wavelength bands with a resolution of 2 arc-second. Our workflow only ran on a small sample taken from this extensive sky survey, showing the Pleiades star cluster. The workflow used for our executions of Montage is presented figure 1.3. It can be understood as 3 Montage sub-DAGs fused together at the last step of the workflow. Each sub-DAG works on the images of a different wavelength band, h, j, and k. These bands are well known for their low rate of absorption by atmospheric gasses and are centered on $1.65\mu m$, $1.25\mu m$ and $2.17\mu m$ respectively. Within the band graphs the project tasks compute the re-projections based on headers contained within the header images. The re-projected images are then sent to an *overlap* task which computes the overlapping regions of the different re-projected images. These overlapping regions are distributed to *diff* tasks charged with comparing the overlapping regions on the re-projected images created by *project* tasks. The comparisons thus created are then used by a *bgmodel* task to compute the necessary brightness and background corrections for every re-projected image. These corrections are applied to every single re-projected image by the *background* task. It should be noted that although it is not the case in our setup, the *background* task can be parallelized in the same fashion as *project* and *diff* tasks. The *add* tasks then splices the different re-projected images in a single mosaic image for the given band. Finally the *gather* task fuses the mosaics produced for each of the bands.

Table 1.2 provides the distributions of input size, runtime, and output size for the different tasks-types across all executions of Montage. It should be noted that *overlap* and *gather* only use as inputs data generated within the workflow. This explains why we found a 0 standard deviation for these input sizes. Conversely *bgmodel* and *gather* outputs are also extremely constant. Overall Montage is much more communication intensive, with data transfer times representing between 42% and 64% of task executions depending on the platform parameters.

Chapter 2

Operating Scientific workloads on IaaS Clouds

2.1 Cloud Scheduling

Our team has been working on the opportunities of using IaaS clouds to execute batch-job like workloads. As defined earlier, clouds are essentially scalable infrastructures. Therefore, executing a workload implies two actions: i) provision an appropriate number of resources, and keeping this number adapted to the intensity of the workload, and ii) map the computations to the resources in a timely manner. Although a small workload could be planned and executed manually by a user, larger workloads will require automation to be properly processed and monitored. Before describing our cloud scheduling system, we quickly define the terms *scaling* and *scheduling*, which correspond to actions i) and ii) respectively, as these terms are frequently encoutered in the literature related to clouds.

2.1.1 Scaling

Scaling is the act of provisioning enough resources for one's workload. Commercial entities and research institutions alike have had to deal with scaling since computers became an essential tool for their activities. The difficulty is striking the balance between acquiring enough resources to fulfill all of their needs without overspending resulting in resources that would sit unused. Cloud computing has not fundamentally changed this core balance at the heart of scaling, but it does offer the user the opportunity to change that balance at any time, as long as they can afford it.

In the prominent field of cloud applications which is web-based applications, scaling is generally the only action that matters. In this field indeed, requests that require processing arrive continuously over time. Therefore, no scheduling of the processing can be anticipated. Given that the requests are independent from one another, scaling is mostly about guaranteeing enough resources are available at any moment to fulfill every request sent to their web service. In such a context a scaling system will usually be concerned with tracking metrics about the overall application load and pro-actively [13,

55] or reactively [15] allow for the adding or removing of resources to the pool. These forms of auto-scaling are available commercially through third party solutions such as RightScale [41] or Scalr [43] or through the cloud operators themselves (e.g. [2]).

The context of finite workload scaling is more about balancing the total runtime of the workload, called the makespan, against a second metric, often the cost or energy requirements. Examples of simple scaling policies designed to minimize the makespan in priority by greedily starting new resources can be found in [53] or [19], while an example of scaling with power consumption minimization can be found in [18].

2.1.2 Scheduling

Scheduling is the act of assigning a task to an available computing resource. Schedulers are at the core in batch-job systems where they were used to split a shared pool of resources between the submitted jobs in a queue. Systems like Condor [51] allowed universities to pool their computing resources into one system that would automatically distribute tasks submitted for execution. Such systems were mainly concerned with the proper repartition of resources and respecting the tasks' constraints, such as dependencies between different tasks. Such schedulers could be divided into two categories:

- **Offline schedulers**: which need to know all the jobs to perform and the available resources ahead of time.
- **Online schedulers**: which accept new jobs on the fly.

In either case schedulers worked within given resources. They could not, as we can in IaaS, provision new resources as needed. The rise of IaaS gave birth to a new class of algorithms combining scaling and scheduling.

2.1.3 Cloud brokers

Cloud brokers are capable of performing both scaling and scheduling in tandem. In some cases these designs were proposed as an extension to a grid scheduler, making it possible to complement local resources by provisioning additional machines from the cloud ([33, 38]). Others fully relied on cloud resources.

PaaS brokers constrain users to their APIs and mask the underlying resources. JCloudScale [28] proposed by Leitner et al. lets the user provide constraints to respect during execution. mOSAIC [39] requires the users to provide the tasks' needs and minimal performance constraints. And Aneka [52] provides a deadline constraint strategy.

IaaS brokers on the other hand provide a more direct access to the underlying resources. IdleCached [49] is a cloud broker that attempts to optimize the cost by scheduling tasks on idle VMs by following an earliest deadline first policy. E-clouds [34] provides offline schedules of a pre-establish list of available applications. The home-grown project Schlouder [35] lets the user select a provisioning and scheduling heuristic from a library containing strategies optimizing for makespan or for cost.

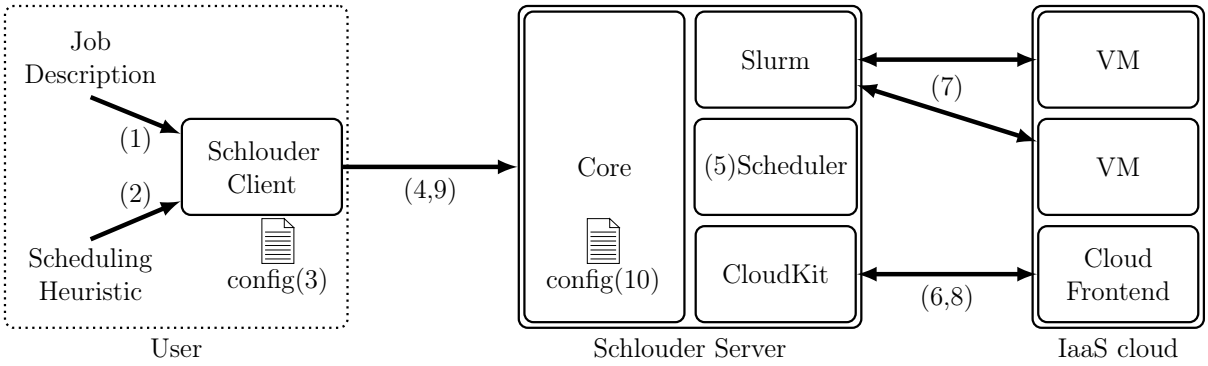

Figure 2.1: The Schlouder cloud brokering system.

2.2 Schlouder

Schlouder, the aforementioned cloud broker, has been developed in our team to schedule batch jobs on IaaS clouds. Schlouder is designed to work with bag-of-tasks and workflows. Schlouder was developed in Perl from 2011 to 2015 by Étienne Michon with contributions from Léo Unbekandt and this author. A functional installation of Schlouder requires a functional installation of Slurm, a job management program service, munge, the authentication service used by Slurm, and a DNS service. Schlouder is made available for usage under GPL licence on INRIA's public forge [45]. This author contributed late to Schlouder's development helping to provide better logging capabilities and to debug scheduling edge-cases.

2.2.1 Schlouder operation

Figure 2.1 offers an overview of Schlouder's operation.

Client-Side operation. The user interacts with Schlouder through the Schlouder client, shown on the left hand side of the figure. To start the execution of a new workload the user provides:

1. *Job description* Provided as a lists of scripts. Each script is a single task. The task's name, expected runtime, and dependencies if any are provided through headers in its script. The rest of the script is executed on the VM when the task is executed.

2. *Scheduling Heuristic* The user chooses from the heuristics installed on the server which one to use to schedule their workload.

The Schlouder *client configuration file* (3) provides additional settings such as the preferred instance type and the cloud selection criterion, when applicable, as well as the address and connection parameters to the Schlouder server.

Workload execution. The client sends all this data to the Schlouder server (4). As the server receives data from the client it immediately starts the process of executing the user's workload.

5. *Scheduling* is done using the heuristic chosen by the user. The scheduler schedules tasks one at a time as soon as all dependencies are satisfied. Tasks are either scheduled to existing VMs or to a VM to be provisioned.

6. *Provisioning* is done through the Cloud Kit module. The cloud kit is the interface between Schlouder and the cloud by implementing the API calls necessary for Schlouder's operation. The cloud kits are API dependent, as such multiple clouds using the same API do not require different cloud kits. The functions required by Schlouder are:

   - `runInstance`: provisions a new instance from the cloud.
   - `instanceIsRunning`: determines whether a given instance has successfully booted and reached the running status.
   - `getIPAddressFromId` & `getHostnameFromId`: these functions are used when pushing tasks to the VMs
   - `describeInstances`: provides a list of all provisioned instances, their current state and IP address when applicable.
   - `terminateInstance`: terminates a specific instance.

   Cloud specific configurations must be provided to the Schlouder server for every cloud currently in use. These files contain the cloud kit to use, cloud connection credentials to use, the VM image to use, a list of available instance types, cloud usage quotas if applicable and a boot time prediction model.

7. *Task execution* Once a VM has been provisioned the tasks queued on it by the scheduler are executed by the task manager, Slurm by default. Slurm copies the task's script to the VM and monitors the scripts execution. When a task finishes Slurm retrieves the execution log before executing the next queued task.

8. *Instance shutdown* is done through cloud kit's `terminateInstance` function. Shutdowns are only performed when a VM is without any running or queued tasks and *it reaches the shutdown margin at the end of its BTU.*

9. *Client monitoring* At anytime during or after the execution the user can use the Schlouder client to monitor the advancement of the execution. The report sent by the server contains the list of all active, shutdown, and planned VMs, and for each of those the list of executed, running, and queued tasks.

Server Configuration. The server configuration(10) contains the necessary information for Schlouder to operate, including:

- *Application setup*: Binding address and port, temporary directory, client usernames, BTU length, and shutdown margin.

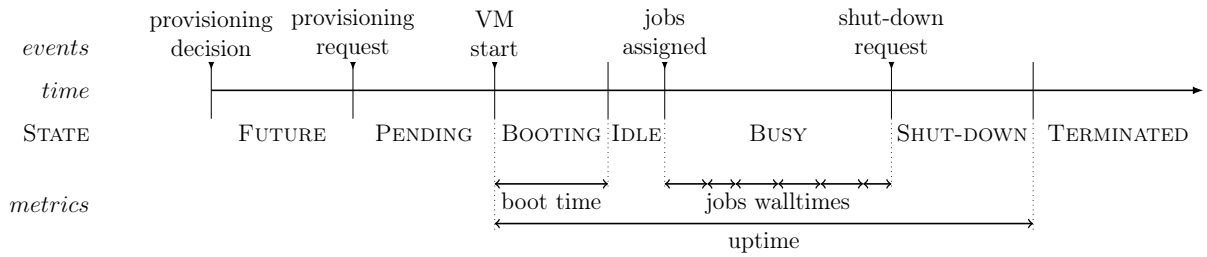

Figure 2.2: VM lifecycle

- *Cloud setup*: List of available clouds and for each cloud :
  - the API to use
  - connection information
  - boot time estimators
  - usage quotas
  - available instance types
  - disk image to use
- *Task manager setup*: system to use and configuration file
- *Scheduler setup*: a list of available scheduling heuristics

Although they are part of the application and cloud setup respectively the BTU length (length of the cloud's billing cycle), the shutdown margin (overhead scheduled for shut-down), cloud usage quotas, and the boot time estimators are also inputs of Schlouder's scheduler.

VM lifecycle management. From the moment the scheduler decides to provision a new VM Schlouder's node manager thread creates the internal representation necessary to track its status. Figure 2.2 schematize this lifecycle, its key event and the metrics reported by Schlouder.

1. Future As soon as Schlouder's scheduler requires a new VM a new task queue is created. The scheduler is made aware of this task queue and immediately able to schedule more tasks to it. The start time/end times of the jobs in the queue are computed on the basis of the boot time estimators and the tasks' expected runtimes. Internally the tasks is assigned to a VM with the Future state.
2. Pending is the state assigned to a VM for which a provisioning request has been submitted to the cloud. As with the Future state, Pending VMs are simple tasks queues existing mostly to allow the scheduler to continue normal operation. The difference between Future and Pending exists for Schlouder to keep track of how many provisioning requests need to be done and how many VMs are still expected from the cloud.

3. BOOTING As soon as a new VM resource appears on the cloud it is assigned to the oldest PENDING VM. The internal representation of the VM is completed with available information from the cloud and its state is switched to BOOTING. Schlouder will store the current time as the *VM's start date* and the task queue is recomputed using this date to allow for more precise scheduling. Two timers are set, the shutdown timer roughly as long as BTU and the short startup monitoring timer. The latter instructs Schlouder to check if the VM has been detected by Slurm. Becoming visible in Slurm switches the VM to the IDLE/BUSY states depending on context. The time elapsed in the BOOTING state is recorded as the *boot time*. The tasks queue is updated with the measured boot time in place of the boot time estimation.

4. IDLE/BUSY During normal operation the VMs will alternate between the IDLE and BUSY state depending on whether they are running any tasks at a given time. Each time a VM finished a given task the task queue is updated to adjust for any deviation for the prediction. This allows the scheduler to always operate on the most precise information available.

5. SHUTDOWN The first shutdown timer of any given VM is setup to trigger one *shutdown margin* (as indicated in the server configuration) before the end of the BTU. Once the shutdown timer is triggered Schlouder checks whether the VM is IDLE and its tasks queue is empty in which case Schlouder will set the VM state to SHUTDOWN and send the corresponding request to the cloud. A VM in the SHUTDOWN state becomes unavailable to the scheduler. VMs which do not meet the shutdown criterion see their shutdown timer extended for 1 BTU.

6. TERMINATED Schlouder monitors the status of VMs in the SHUTDOWN state until they are removed from the cloud resource list. When this happens the VM's state switches to TERMINATED and the current time is stored as the *VM's stop date*. The VM's *uptime* is computed from the VM's start date and stop date and added to Schlouder's output.

Task lifecycle management. Like with node management Schlouder also keeps a task manager thread in charge of tracking the different tasks lifecycle and stores many metrics for review and analysis. The lifecyle of a single task is presented figure 2.3.

1. PENDING Tasks received from a client are added to the scheduling queue. The time at which the server receives the tasks is called the *submission_ date*. Newly received tasks are attributed the PENDING state.

2. SCHEDULED Once all the dependencies of the task reach the COMPLETE state the task can be scheduled. Once the scheduler assigns the tasks to a task queue the task's state is switched to SCHEDULED. The time at which the scheduling happens is recorded as *scheduling_ date*.

3. SUBMITTED A task will remain in its SCHEDULED state until its turn to be executed on its assigned node comes up. Once the task is handed over to Slurm for execution

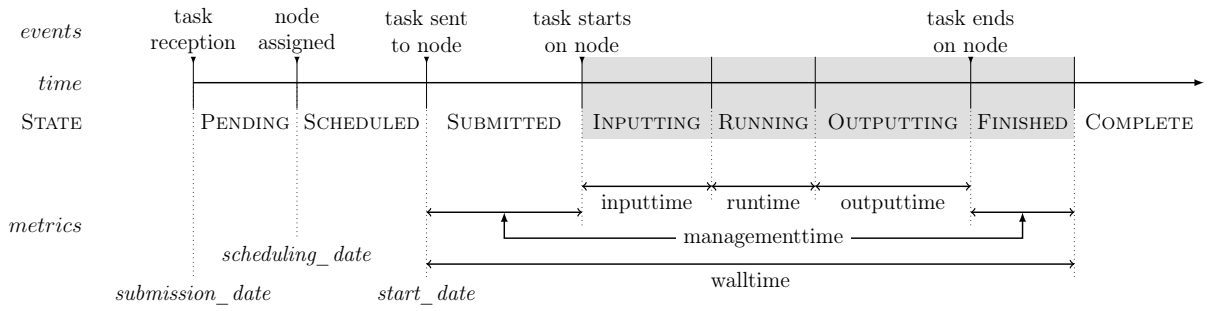

Figure 2.3: Tasks lifecycle

the task is considered Submitted and the current time recorded as the task's *start_ date*.

4. Once the task is under Slurm's control Schlouder does very little to track its start and its advancement. However for tasks following the intended format it is possible to report to Schlouder an *input_ time*, *runtime*, and *output_ time*. These times are retrieved from the task's execution log and therefore not known by Schlouder. As such the following states exist for research purposes and not as part of the Schlouder internal representation.

   - Inputting The task starts executing on the node, and downloads from storage the inputs necessary for its execution.
   - Running The task executes its main payload on the inputs.
   - Outputting Results from the execution are uploaded to storage.
   - Finished The task's script has finished running on the node. But Schlouder has yet to become aware.

5. Complete Schlouder becomes aware that the task has finished. The current time is used in conjunction with the *start_ date* to compute the jobs *effective walltime*. The task state is set to Complete and the task queue to which it belonged is updated to account for the effective walltime.

All the Schlouder metrics described in the previous paragraphs are defined in table 2.1. Times and dates are provided in seconds and recent versions of Schlouder have a precision of 1 nanosecond.

Although we just explained the general operation of Schlouder, the most important component during the execution of a workflow is the scheduler. This is the component responsible for the ordering the provisioning of new VMs and the placement of tasks.

2.2.2 Scheduling heuristics

Schlouder's scheduler takes scheduling and provisioning decisions on a task by task basis, scheduling every task with no pending dependencies. Scheduling decisions are informed

Target	Name	Unit	Source
VM	*boot_time_prediction*	*s*	computed from configuration.
	boot_time	*s*	measured during BOOTING state.
	start_date	date	logged at the start of BOOTING state.
	stop_date_prediction	date	based on *start_date*, BTU length, and task predicted/effective walltime.
	stop_date	date	logged at the start of TERMINATED state.
Task	*submission_date*	date	time at which Schlouder first receives a task.
	scheduled_date	date	time at which the scheduler selects a VM for the task.
	start_date_prediction	date	time at which the scheduler expects the task to start.
	start_date	date	time at which a task is handed to Slurm for execution.
	walltime_prediction	*s*	provided by the user.
	(effective) walltime	*s*	measured by Schlouder.
	input_time	*s*	`provided by the task,` the time spent downloading inputs.
	input_size	byte	*provided by the task,* the size in bytes of different downloaded inputs.
	runtime	*s*	*provided by the task,* the time spent in computations.
	output_time	*s*	*provided by the task,* the time spent uploading results.
	output_size	byte	*provided by the task,* the size in bytes of the different uploaded outputs.
	management_time	*s*	*Computed by subtracting runtime, input_time, and output_time from the walltime.* The management_time represents unmeasurable delays such as the task's transfer time and the delay between the tasks ending and Schlouder being aware of the termination.

Table 2.1: Schlouder available metrics. All times and date measured in seconds with a precision to the nanosecond.

by the scheduled task's expected runtime, the active VMs and their runtimes, eventual future VMs and their expected boot-times, and the expected runtimes of jobs already queued to active VMs.

Alogrithm 2.1 (see below) presents the generic algorithm used for scheduling in Schlouder. This algorithm uses the following definitions:

- t the task to schedule.
- v an available VM
- V the set of all such VMs
- C the set of candidate VMs for scheduling

The scheduler first determines which available VMs can be used to run the task to schedule. This is done using a heuristic-dependent eligibility test, *eligible()*. Then if no VM matches the eligibility conditions a new VM is provisioned and the tasks are queued to it. In cases where one or more VMs are deemed eligible a heuristic-dependent *select()* function is used to select the VM on which to queue the tasks t.

Algorithm 2.1 Generic Schlouder scheduling algorithm

```
1:  procedure SCHEDULE(t)          //a new task t is submitted
2:      C ← ∅                      //C is the set of candidate VMs (C ⊂ V)
3:      for v ∈ V do
4:          if eligible(v, t) then //Find eligible VMs
5:              C ← C ∪ {v}
6:          end if
7:      end for
8:      if C ≠ ∅ then
9:          v ← select(C)          //Select VM amongst eligible
10:     else
11:         v ← deploy()           //Create and run a new VM
12:         V ← V ∪ {v}
13:     end if
14:     enqueue(v, t)              //Map the job to the VM
15: end procedure
```

Schlouder is provided with a dozen strategies derived from this generic setup. *1VM4All* only ever provisions one VM and represents the worse case scenario in terms of makespan while offering the lower cost of execution. At the opposite end of the spectrum *1VMper-Task* immediately schedules every single task on a new VM, therefor offering the shortest possible makespan while being the most expensive heuristic. In this work we will observe Schlouder executions using two heuristics, *As Full As Possible* (AFAP) and *As Soon As Possible* (ASAP).

The AFAP strategy aims to minimize the cost of running a workload. As such it will favor delaying tasks to execute them on already running VMs. However since running

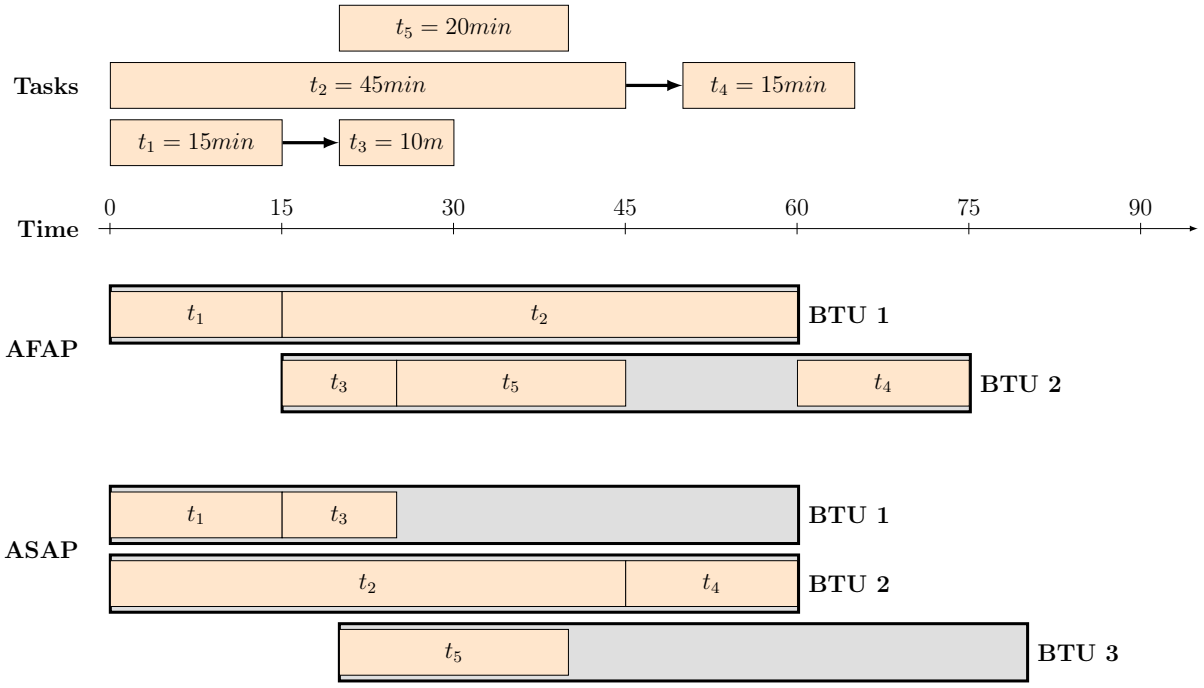

Figure 2.4: Chronograph of the two Schlouder strategies used in this work.

two VMs in parallel has the same cost as running a single VM for two BTUs, AFAP will only consider eligible for scheduling VMs whose expected remaining time can fit the full expected runtime for the task being scheduled. In cases where multiple VMs exists, AFAP will select the *best fit*.

The ASAP strategy aims to minimize the makespan of a workload. As such it will favor provisioning new VMs over delaying a task's execution. Therefore it will only consider eligible for scheduling VMs that are already idle at the time of scheduling and VMs that are expected to become idle before an new one can be provisioned. If multiple such VMs exists, ASAP will select *best fit*.

The expected remaining time of a VM, used in AFAP, is computed using the observed runtime of already executed jobs and the user provided expected runtime of already queued jobs. The expected boot time of a VM is predicted using the parameters provided in the cloud configuration files. These parameters must be estimated, from experience by the operator of the server, and take into account the number of VMs already booting.

Figure 2.4 presents a short example of ASAP and AFAP in a 5-task workload. Tasks t_1 to t_2 are submitted at the start of the workload with tasks t_3 and t_4 being dependent on t_1 and t_2 respectively. Task t_5 is submitted separately 20 minutes after the other tasks.

Chapter 3

Simulation of IaaS Clouds

We now have a scheduling tool to execute our workload in the cloud. Experimentation with scheduling shows that for users to choose the correct schedule to fulfill their constraints is a non-trivial problem. As such, a prediction tool is needed for the user to project the cost and makespan of their workload. We intend to use simulation to provide this prediction.

Simulation has always been a key component of the study of distributed systems. Simulations present three main advantages over experimentations on real platforms. Firstly distributed systems present high installation and maintenance costs, which simulators do not have. Secondly simulations can run experiments faster than real time, allowing to perform lengthy experiments in reasonable times. Lastly, well designed simulations offer an access to the minute-to-minute evolution of systems that are complicated to reproduce, albeit not impossible. However these advantages come at the risk of having the simulation not represent reality. As a gap between the simulation and the corresponding reality is inevitable, if we intend to use simulation for prediction tools we need to establish the deviation between our simulations and reality.

In this chapter we first explore the pervasive principles used in cloud simulation and the available simulation tools. Then we will present the technical solution we used to create a simulator for Schlouder. Finally we will present the evaluation we performed for our simulator and discuss the results' implications on the use of simulation as a prediction tool.

3.1 Simulation technologies

Discrete event simulations. Most cloud simulators are based on discrete event simulation (DES). A discrete event simulation is a series of events changing the state of the simulated system. For instance, events can be the start and end of computations or of communications. The simulator will jump from one event to the next, updating the times of upcoming events to reflect the state change in the simulation. A solver, at the core of the DES, considers the system's states generated by the platform and previous events to

compute the timing of future events. In most cases, simulators have a *bottom-up* approach: the modeling concerns low-level components (machines, networks, storage devices), and from their interactions emerge the high-level behaviours. Working on disjoint low-level components makes it easier to tune the model's precision to the wanted accuracy or speed trade-off.

Running a bottom-up DES-based simulator requires at least a platform specification and an application description. The platform specification describes both the physical nature of the cloud, e.g. machines and networks, and the management rules, e.g. VM placement and availability. Depending on the simulator, the platform specification can be done through user code, as in CloudSim [10] for example, or through platform description files, as is mostly the case in SimGrid [14]. The application description consists in a set of computing and communicating jobs, often described as an amount of computation or communication to perform. The simulator computes their duration based on the platform specification and its CPU and network models. An alternative approach is to directly input the job durations extrapolated from actual execution traces.

The available cloud DESs can be divided in two categories. In the first category are the simulators dedicated to study the clouds from the provider point-of-view, whose purpose is to help in evaluating the datacenter's design decisions. Examples of such simulators are MDCSim [30], which offers specific and precise models for low-level components including the network (e.g. InfiniBand or Gigabit ethernet), operating system kernel, and disks. It also offers a model for energy consumption. However, the cloud client activity that can be modeled is restricted to web-servers, application-servers, or data-base applications. GreenCloud [24] follows the same purpose with a strong focus on energy consumption of the cloud's network apparatus using a packet-level simulation for network communications (NS2). In the second category are the simulators targeting the whole cloud ecosystem, including client activity. In this category, CloudSim [10] (originally stemming from GridSim) is the most broadly used simulator in academic research. It offers simplified models regarding network communications, CPU, or disks. However, it is easily extensible and serves as the underlying simulation engine in a number of projects (e.g. ElasticSim). Simgrid [14] is the other long-standing project, which when used in conjunction with the SchIaaS cloud interface provides similar functionalities as CloudSim. Among the other related projects, are iCanCloud [36] proposed to address scalability issues encountered with CloudSim (written in Java) for the simulation of large use-cases. Most recently, PICS [23] has been proposed to specifically evaluate simulation of public clouds. The configuration of the simulator uses only parameters that can be measured by the cloud client, namely inbound and outbound network bandwidths, average CPU power, VM boot times, and scale-in/scale-out policies. The data center is therefore seen as a black box, for which no detailed description of the hardware setting is required. The validation study of PICS under a variety of use cases has nonetheless shown accurate predictions.

Alternative approaches to simulations. Although the vast majority of cloud simulators are based on *bottom-up* DESs other approaches are available for creating simulations. One such approach is directed acyclic graph (DAG) computation. This kind of approach is extensively used in project management where methods such as PERT (*Program Evaluation and Review Technique*) or Gantt graphs are commonly used to estimate project

length. This method works well with the simulation of the types of workflows studied in this work, where tasks have clear ends and defined dependencies, but would not be appropriate for generic cloud simulations. It should be noted that this kind of approach does not preclude the use of a DES, but using PERT methodologies allows one to perform the same simulation from a workflow perspective where the intricacies of the underlying resources are not formalised. Still others build simulators using a *top-down* approach. *Using Trustworthy Simulation to Engineer Cloud Schedulers* [40] describes the creation of a simulation model based on perturbation theory. Starting with the most parsimonious model possible, the authors of this work introduce a perturbation in the inputs to represent unmodeled behaviors. If the simulation is insufficiently accurate, terms are added to the main model and the perturbation adjusted in consequence. It should be noted that in this work the simulator using the thus obtained model keeps track of the simulation state using a DES. The originality of this work compared to the ones described in the previous paragraph lies in the use of a *top-down* model and the use of a perturbation term covering unmodeled behavior.

3.2 Building a Simulator

We designed SimSchlouder as an aptly named simulator for Schlouder executions. SimSchlouder is based on SimGrid and uses SchIaaS, a specifically built IaaS simulation extension for SimGrid.

3.2.1 SimGrid

SimGrid [14] is a scientific instrument to study the behaviour of large-scale distributed systems such as Grids, Clouds, HPC, and P2P systems. The SimGrid project was started in 1999 and is still under active development [46].

SimGrid Engine. Simgrid itself is not a simulator but rather a framework and toolkit with which to build DESs. SimGrid provides core functionalities needed for a simulator and a number of user APIs with which to interact with the core functions. Figure 3.1 schematizes the internal structure of the SimGrid toolkit. The key components are:

- **XBT** contains all the base data-structures and memory handling functions. Written in C this module helps guarantee the portability of SimGrid-based simulators.
- **SURF** is the solver at the core of the SimGrid simulations. Using the platform representation this module computes the timing of events within the DES.
- **SIMIX** is a kernel handling the usercode and acts as an application description. The processes handled by SIMIX are executed or suspended in accordance with the timings and events computed by **SURF**.
- *User APIs* The remaining modules represent the APIs used by the user creating a simulation based on SimGrid.

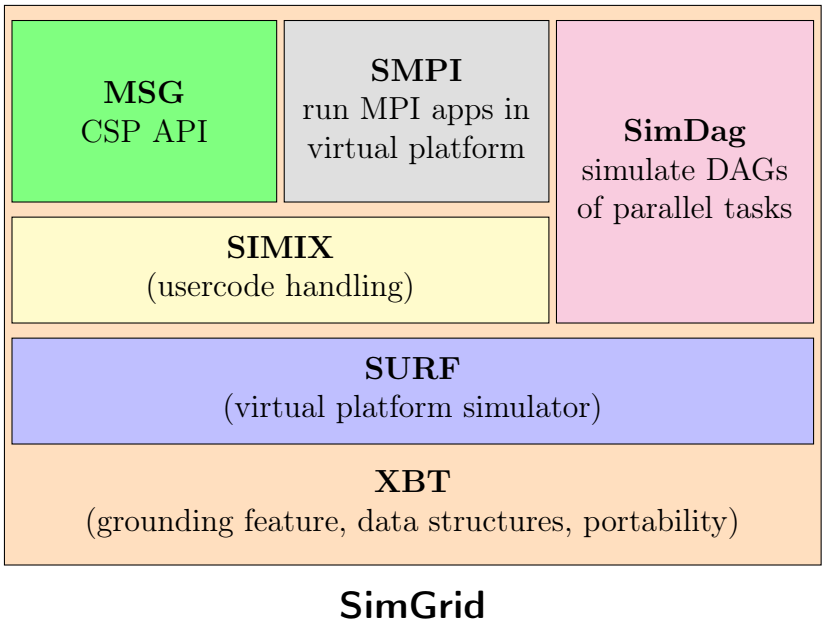

Figure 3.1: SimGrid Architecture

- **MSG** allows the user to represent the simulation content to be presented as *Communicating Sequential Processes* (CSP). When using MSG the usercode describes a sequence of communications and computations performed by the simulated process.
- **SMPI** also simulates the application as a set of concurrent processes, but the simulated processes are generated automatically from an existing C or Fortran MPI enabled application. SMPI provides an additional runtime for MPI-specific functionalities.
- **SimDAG** does not use concurrent processes as a representation of the simulated workload but instead allows for an abstract representation as a directed acyclic graph of communicating computational tasks.

Simulations using MSG. Our work relies on the **MSG** API. This API is available in C, C++, Java, Ruby, and Lua. Writing a simulator with MSG requires the user to provide:

- **Simulator code** as SimGrid is merely a framework. The user must write a code to instantiate the SimGrid engine and configure it properly before starting the simulation itself.
- **Platform file** provides a description of the simulated environment within which the simulation happens. It mainly consists in a description of available machines, called hosts, and the networks to which they are connected. SimGrid's platform syntax offers a wide array of options such as cluster, CPU speed, CPU power consumption, network bandwidth and latency. A number of parameters can be replaced by files

that allow the user to vary the parameters along the simulation, e.g. allowing for a simulation where the bandwidth of a link changes over time.

- **Usercode** representing the actions undertaken by the simulated processes. This code is compiled with the simulator code and will be executed during the simulation at the will of the SIMIX kernel. The usercode uses specific MSG function calls to indicate to the simulator that the simulated process is performing a specific action such as communicating or computing a certain amount of data. Using code to represent the simulated application allows the simulation to react to simulated events in a way static representations can not. In our case usercode allows us to implement task scheduling within the simulation.

- **Deployment file** indicates to SimGrid on which host of the simulated platform the different simulated processes are executed. This is optional as pinning processes to a host can also be done in the simulator code directly.

A well coded simulator first initialises the SimGrid engine with the MSG user API. Second, the platform file is passed to the engine. Third, simulated processes are set up, either through manual calls to MSG or by passing the deployment file. In this step SIMIX acquires pointers to the usercode to execute during the simulation. Finally the MSG_run function is called to start the simulation.

During the simulation code execution is controlled by the SIMIX kernel. While the simulated processes are running the simulation's clock is suspended. A simulated process is suspended when it makes an MSG call and the event corresponding to that call is added to the simulation's timeline. Once all the processes are suspended SURF solves the timing of all the events on the timeline. The simulation's clock in then advanced to the next event on the timeline and the process suspended by the corresponding call is awaken. The simulator continues alternating between the execution of usercode and the computation of the timeline until all usercode is terminated. At which point the simulation ends and control is handed back to the simulator code.

IaaS cloud within SimGrid. SimGrid is a lower level toolkit to build simulations of distributed systems. Its *bottom-up* design means SimGrid concerns itself with hosts, their cores, their network interfaces, and the sharing of those resources between the processes working on the hosts. Because of this SimGrid by itself offers very little in terms of tools for cloud simulation. It does however offer one critical element, the ability to partition a host's resources into simulated VMs. From a functionality perspective the VM interfaces available in MSG matches the functionality expected of an hypervisor kernel module. That is the ability to create of VM based on a number of available physical resources, the ability to start suspend or shutdown this simulated VM, and the ability to assign simulated processes to the VM instead of the underlying host. SimGrid VMs are heavily abstracted and do not have boot or shutdown times. In order to build our simulation of Schlouder we needed to extend the base provided by SimGrid into a proper cloud simulation.

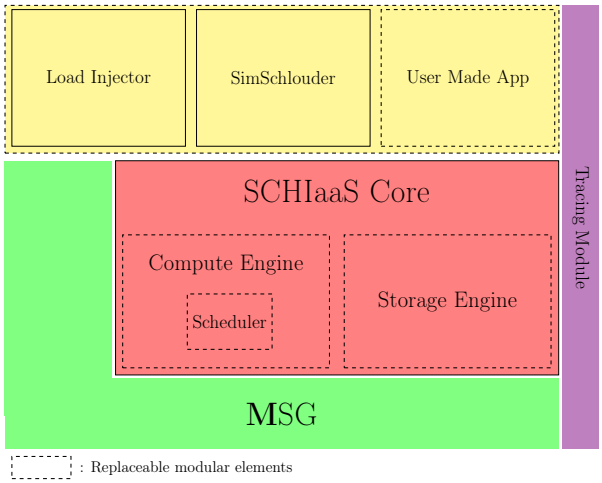

Figure 3.2: SchIaaS Architecture

3.2.2 SchIaaS

We designed SchIaaS to provide a tool to build extendable IaaS simulations. SchIaaS is written in Java and works with the Java MSG API. SchIaaS was designed and written as part the ANR project SONGS (*Simulation Of Next Generation Systems*) of which SimGrid was also part and is available at [44]. Like SimGrid, SchIaaS is designed as a toolkit that is not a simulator by itself but an extension of the of the calls available in MSG. SchIaaS is not however a patch to SimGrid, it rests on top of MSG and SchIaaS call modify the simulation state by a series of coordinated calls to the MSG interface.

Structure. SchIaaS was designed to facilitate both simulations of IaaS clouds and simulations of applications using IaaS clouds. Figure 3.2 presents an overview of the structure of SchIaaS. Key elements are as follows:

- **SchIaaS core** contains the base code for initialisation and loading of modules.
- **Compute Engine** this component contains the code pertaining to VM lifecycle management. Although a compute engine is provided with the base installation of SchIaaS, this module can be replaced for experiments not compatible with the base implementation.
  - **Scheduler** this component determines on which hosts the VMs are placed on. Because this component is often the subject of independent studies this element can be swapped independently from the standard Compute engine. The SchIaaS installation provides a couple of different schedulers.
- **Storage engine** this component provides a bucket storage element, with the ability to *put* and *get* data from a storage service. This element is replaceable to allow for experimentation with data management.

Additionally SchIaaS contains two optional modules:

- *Tracing module* designed to help users extract useful information from the simulated environment.

- *Load injector* performs simulated VM startup and shutdown requests following user provided pattern or traces. This module is useful for users trying to study behaviours of the cloud itself rather than an application using the cloud.

The last usercode presented in Figure3.2 is our simulator SimSchlouder described hereafter.

Functionality. The functions provided by SchIaaS mirrors what is expected of a cloud API. Where SimGrid provides very basic hypervisor level functionalities to control the simulations of VMs, SchIaaS deals in cloud instances. Using SchIaaS the simulated processes can query available images and instance types, and run, suspend, resume, or terminate instances of the required type. Upon request, SchIaaS will provide the underlying simulated VM to the usercode to allowfor the allocation of new simulated processes. Additionally the storage interface allows the simulated processes to trigger uploads and downloads from the dedicated storage nodes.

Usage. To use SchIaaS within a simulation the user must initialise the SchIaaS component during startup after initialising MSG and loading the platforms. Initialisation is configured by a cloud file. This file describes the simulated clouds. For each cloud the file indicates the Storage and Compute engine, the scheduler, the compute nodes, and the available flavors and disk images. During the simulation simulated processes can make calls to SchIaaS engine functions. These functions are executed as usercode but themselves make calls to the appropriate MSG functions triggering the events corresponding to the operations of the simulated IaaS cloud.

3.2.3 SimSchlouder

SimSchlouder is a simulation of Schlouder built on top of SimGrid and SchIaaS. It can be found in the SchIaaS repository ([47]). SimSchlouder was first designed as a prediction module for Schlouder, with the objective of offering to the users a preview of the makespan and costs to expect depending on their chosen strategy. Although a prediction module would have to strictly use the same inputs as Schlouder, we quickly extended SimSchlouder to be a general purpose Schlouder simulator with multiple possible levels of inputs. This evolution has helped us make SimSchlouder more accurate, as we will see in Section 3.3, and on occasions has lead finding bugs in Schlouder itself.

Usage. SimSchlouder runs a simulation of a given platform with a specified cloud. The simulation encompasses the target cloud with its physical components and the Schlouder server with its physical host. The Schlouder-client and its communication with the Schlouder-server are not part of the scope of the simulation. In a run of SimSchlouder

only one strategy may be used, however the simulation allows for jobs to arrive in separate batches and at different times during the simulations. SimSchlouder's invocation is as follows:

```
$java simschlouder.SimSchlouder simschlouder.xml job.tasks strategy [flags]
```

- `SimSchlouder.xml` is the main configuration file. This file contains a mix of parameters necessary for a SimGrid/SchIaaS simulation and parameters found in Schlouder's configuration.
  - the platform file to use for the simulation. (necessary for SimGrid);
  - the cloud configuration file to use with SchIaaS;
  - the simulated host on which the simulated Schlouder-server is executed (necessary to setup the simulation);
  - the clouds to use (as in Schlouder's configuration);
  - the BTU length for each cloud (as in Schlouder's configuration);
  - the boot time estimators to use (as in Schlouder's configuration);
  - the shutdown margin to use (as in Schlouder's configuration);
  - the image and instance type to use on each cloud (as in Schlouder's configuration).
- `job.tasks` contains the description of the tasks to execute. As in Schlouder this list must include for each task a name, a predicted runtime, and data dependencies. Since client-server communication falls outside of the simulation's scope SimSchlouder additionally requires that each task be assigned a submission date. Moreover any input necessary for the advanced simulation presented in the *Capabilities* paragraph below will also be added to this file.
- `strategy` The heuristic to use for provisioning and scheduling within the simulation. The strategies match the ones found in Schlouder, but differences in programming language mean that strategies had to be reprogrammed in SimSchlouder.
- `flags` indicates which optional fields from the task file must be used during the simulation.

Capabilities. The most basic use-case of SimSchlouder is the prediction module setup limited to the inputs provided to Schlouder. This strongly reduces the simulation accuracy. To tune the simulation, the simulator was extended to take into account additional timing inputs, triggered by using the following flags:

- *real_boottimes*: SimSchlouder can receive real boot times as additional input. Real boottimes create a delay in the start of the first task of their respective nodes. This option mostly affects the simulated makespan, and possibly later scheduling decisions. Without *real_boottimes* the simulation will make use of the estimators provided in the configuration file.

- *real_ walltimes*: In the basic use-case SimSchlouder considers the user to be correct in his walltime predictions and uses them not only for the scheduling but also as runtimes during the simulation. Of course, walltime predictions being exact is an exceedingly rare case. Submitting real walltimes allows the user to create realistic simulations where predictions do not match the effective walltimes.

- *communications*: For users interested in the impact of communications on their workload, this setup allows them to specify a computation time and the input and output data sizes instead of the task's walltime. This kind of simulation can be helpful if the user expects network contentions to be a big contributor to transfer times.

- *real_ threads*: Because of Schlouder's threaded structure there remains some level of unpredictability as to when some event happens. Chief amongst these is the timing of scheduling. This option allows the user to specify for each task the *scheduling_ date* at which the scheduler will consider planning the tasks.

To measure the impact of these capabilities on simulation accuracy we build an automated validation system.

3.3 Evaluation of SimSchlouder

Over the development of Schlouder we accumulated 336 execution traces of Montage and OMSSA in two cloud environments named openstack-icps and BonFIRE. We use these traces as a reference point against which to test SimSchlouder.

3.3.1 Experiment rationale

This validation experiment's primary goal is to check whether SimSchlouder operates correctly to evaluate the gap between simulation and reality. Additionally we intend to measure the influence of additional data, as introduced in the capabilities list in on page 28, on the accuracy of the simulation.

This experiment, presented in figure 3.3, uses one-to-one simulations of real executions. In a simulation *batch*, every real execution in our archive gets simulated once. Results from the simulation are then compared to the real execution on which the simulation was based.

This processed is repeated for each batch with different combinations of SimSchlouder capabilities. This allows us to evaluate how the data provided by each capability impacts the simulation accuracy. The simulations from the batch called *best* uses `real_boottimes`, `real_walltime`, and `real_threads`. This removes any guesswork from the simulated *boottimes*, *walltime*, and *scheduling dates*, and insures that the remaining errors can be fully imparted to the simulator. Conversely, the *blind* batch only uses the predicted walltimes provided by the user. This batch produces the prediction SimSchlouder would give

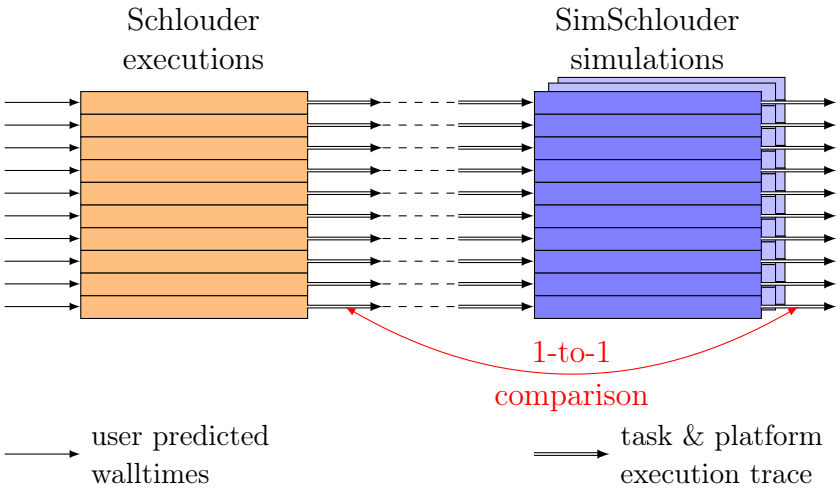

Figure 3.3: Representation of the evaluation experiment. 1. Traces from a real execution are injected in a SimSchlouder simulations. 2. The simulations results are compared to the traces from the corresponding real execution.

before an execution. Additional batches providing only some of SimSchlouder capabilities are used to study the impacts of real timing data on the simulation accuracy.

Through this experiment we intend to comprehensively measure the gap between simulations and reality and which parameters help simulation precision the most.

3.3.2 Simulation Tooling

This evaluation raised a technical difficulty as we needed to repeatedly run 274 simulations, each based on a different execution trace. Since doing so manually would certainly lead to omissions or mistakes, we designed a simulation manager capable of automating the execution of a large number of simulations described in a short configuration file. This tool called `lab.py` is a key component of this SimSchlouder's evaluation and of the work presented in part two of this thesis. The lab.py can also be found in the SchIaaS repository at [44].

The lab.py is a python script used to control repeated executions of an application. The script takes as input a configuration file. The configuration used in the evaluation experiment is available as Configuration 3.1. The key elements of the configuration are:

- `PRE_COMMAND_SETUP`
  A shell command to be executed before any simulation. The command presented here calls a script that generates a list of simulation inputs in a file called `simulations.cfg`.

- `SIM_ARG position[:name] arg [args [...]]`
  The `SIM_ARG` indicates one or multiple arguments to pass to the simulator. Each `SIM_ARG` call includes a `position`. The lab will execute as many simulations as there are combinations of `SIM_ARG`s respecting the proper order of positions. The

Configuration 3.1 `validation.cfg`, the lab configuration file used for evaluation of SimSchlouder.

```
SETUP_DIR ./setup/simschlouder
NEEDED ./xml/*
PRE_COMMAND_SETUP ./validation-pre.sh tasks/* > /tmp/simulations.cfg
POST_COMMAND_SETUP ./validation-post.sh > results/metrics.dat

SIM_ARG 1 simschlouder.SimSchlouder

INCLUDE /tmp/simulations.cfg

SIM_ARG 3:blind
SIM_ARG 3:best real_walltimes real_boottimes real_threads
SIM_ARG 3:no-walltimes real_boottimes real_threads
SIM_ARG 3:no-threads real_boottimes real_walltimes
SIM_ARG 3:no-boottimes real_walltimes real_threads
```

optional `name` is used to differentiate between simulations based on the arguments used.

- `INCLUDE`
  takes the content of a file and adds it to the current configuration. In the case presented here `simulations.cfg` was generated by the pre-command and contains a list of `SIM_ARG`s, with 2 as a positional argument, each corresponding to a real execution trace in our archive. These `SIM_ARG`s indicate the SimSchlouder configuration, the task file, and the strategy to use.

With this simple configuration file we use the `validation-pre.sh` to generate the task files and SimSchlouder configuration for every single execution present in our archive. Each of these simulations, listed in `simulations.cfg`, are then simulated 5 different times corresponding to the 5 batch types described by the `SIM_ARG`s for position 3:

1. the *blind* batch uses none of SimSchlouder extended capabilities. These simulations reproduce the results obtained by the prediction module of Schlouder.

2. the *best* batch uses the `real_walltimes`, `real_boottimes`, and `real_threads` flags. These simulations show the best precision the simulation can achieve.

3. the *no-walltimes* batch uses the `real_boottimes` and `real_threads` flags. By comparison with the simulations from the *best* batch, these simulations measure the impact of using real walltimes on the simulator's accuracy.

4. the *no-boottimes* batch uses the `real_walltimes` and `real_threads` flags. These simulations are used to measure the effectiveness of using real boottimes in our simulations.

5. the *no-threads* batch uses the `real_walltimes` and `real_boottimes` flags. These simulations measure the impact of real scheduling dates on the simulator's accuracy.

Once all the simulations have been performed, the `validation-post.sh` setup as a *post command* will aggregate the results and compute the resulting metrics.

A single call to `lab.py` with this configuration triggers the execution of 1370 simulations. All of the simulations are fully managed by `lab.py`. Should the simulations happen on a multicore system, lab.py allows the user to specify the number of simulation to execute in parallel to shorten the overall execution time.

3.3.3 Analysis

The `validation-post.sh` script automates a large part of the validation by extracting the metrics from the simulation and real execution traces, and comparing them on a one-to-one basis. The compared metrics are:

- *makespan*: the time elapsed between the submission of the first task and the end of the last task. Represents the time elapsed from the user's point of view.
- *schederror*: the number of tasks that are not assigned to the same node in the simulation compared to the reality. Represents the accuracy of the scheduling decisions.
- *uptime*: the amount of resources rented in seconds of available VM time. This metric is an analog to the cost of the execution.
- *usage*: the ratio of the sum of all task walltimes over uptime. Represents the efficiency of the provisioning.

To normalise results across all applications, clouds, and strategies, we compute the absolute error on every metric for every simulation:

$$m.ae = \frac{|m^S - m^R|}{m^R}$$

with $m.ae$ the absolute error on the metric m, m^S and m^R the values of the metric in a simulation, and its corresponding execution trace respectively.

3.3.4 Results

We now have all the information necessary to measure the precision of SimSchlouder, and the impact of the available inputs on the results. Results are split by cloud of origin, with our archive containing traces from executions on our local cloud (openstack-icps) and execution on a European experimental BonFIRE cloud platform ([7]).

Figure 3.4 presents the repartition of absolute errors across all metrics and all simulation types presented in Section 3.3. On boxplots the box spans from the first quartile to the third quartile, with the median represented as a bar across the box. The whiskers will extend to cover any outliers up to 1.5 times the inter-quartile range. Any outliers beyond that are represented by individual points. Corresponding data is reported in Table 3.1. The table provides the values of the first quartile, the median, and the third quartile for the absolute error on every metric and all simulations types.

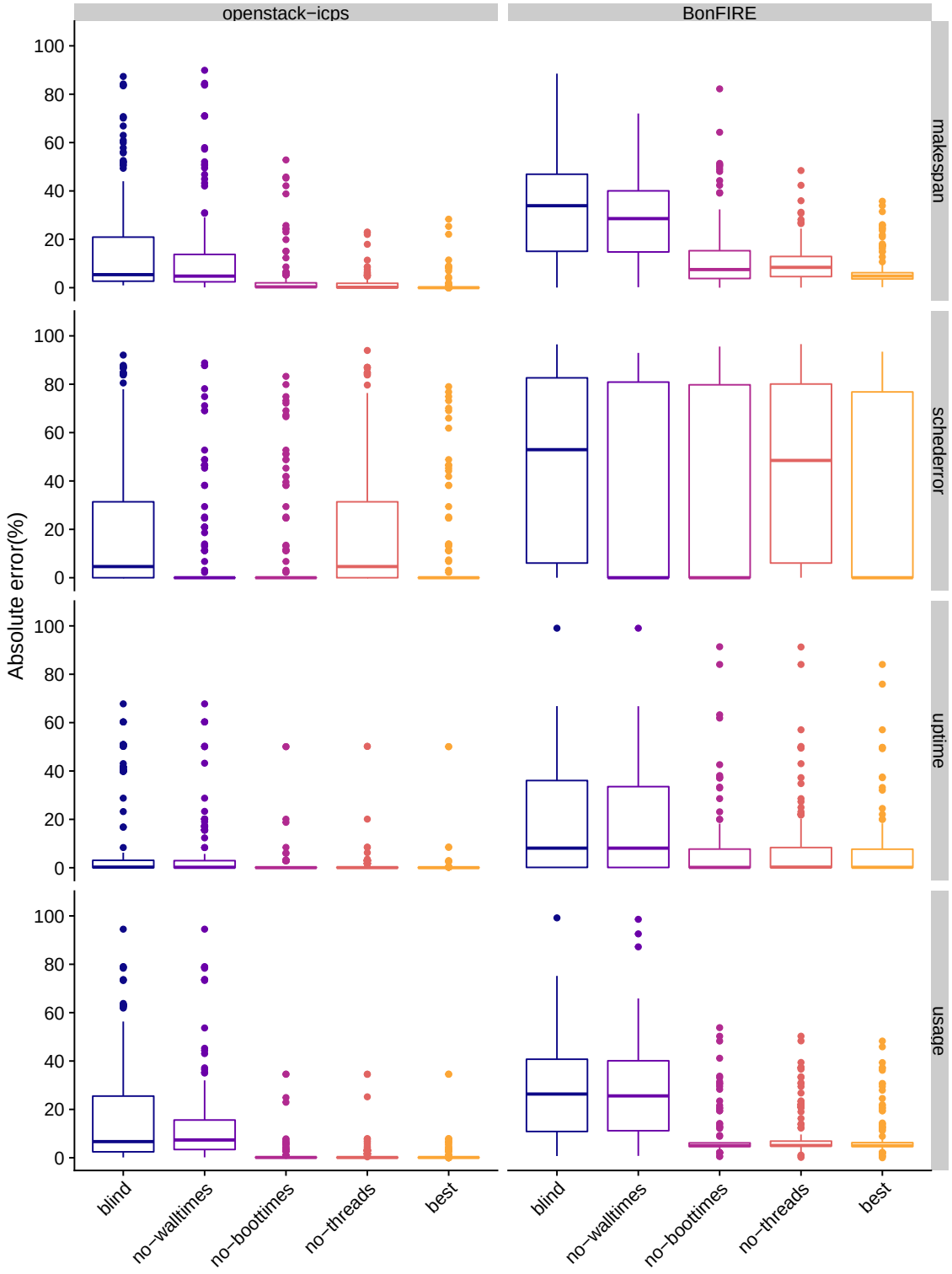

Figure 3.4: Repartition of the absolute error for every application, every metric, and every simulation setup. The boxes represent the first to third quartile range. The horizontal crossbar inside the box represents the median. The whiskers represent values within 1.5 inter-quartile range of the box, with values further away being represented by points. Due to space constraints, 13 outliers, out of 7320 values, presenting an absolute error over 100% have been excluded from the graph.

openstack-icps	makespan.ae (%)			schederror.ae (%)			uptime.ae (%)			usage.ae (%)		
blind	2.69	5.42	21.15	0.00	4.61	31.39	0.03	0.25	3.06	2.45	6.88	25.82
no-walltimes	2.44	4.80	13.88	0.00	0.00	0.00	0.03	0.18	2.93	3.42	7.43	15.74
no-boottimes	0.21	0.33	2.01	0.00	0.00	0.00	0.02	0.03	0.18	0.08	0.12	0.23
no-threads	0.01	0.17	1.82	0.00	4.61	31.39	0.01	0.04	0.18	0.07	0.11	0.18
best	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.10	0.08	0.12	0.13
BonFIRE	makespan.ae (%)			schederror.ae (%)			uptime.ae (%)			usage.ae (%)		
blind	15.04	33.89	46.91	6.06	52.94	82.63	0.14	8.13	36.06	11.05	26.60	42.56
no-walltimes	14.77	28.56	40.03	0.00	0.00	80.88	0.11	8.10	33.57	11.19	25.68	40.33
no-boottimes	3.79	7.49	15.29	0.00	0.00	79.76	0.07	0.14	7.71	4.66	5.01	6.17
no-threads	4.59	8.41	12.91	6.06	48.48	80.08	0.10	0.25	8.35	4.85	5.07	6.88
best	3.60	4.81	6.20	0.00	0.00	76.81	0.08	0.18	7.67	4.66	5.01	6.27

Table 3.1: Repartition of the absolute error on the makespan, scheduling errors, uptime, and usage for all the simulations types presented in Section 3.3.2. Each column follows a *first quartile median third quartile* format.
Reading example : On the openstack-icps platform the blind batch has a median absolute error on makespan of 5.42%, 25% of simulations have an error $\leq$ *2.69% and 75% of simulations have an error* $\leq$ *21.15%.*

The *blind* simulation batch. The first batch of simulations is performed without any additional input. These simulations are the ones that would be obtained from running SimSchlouder as part of Schlouder's prediction module. The quality of these results depend on the accuracy of the expected walltimes provided to Schlouder and SimSchlouder. These expected walltimes are usually obtained through user experience. In our case, these were obtained through performing manual executions of tasks on the openstack-icps platform, early in the Schlouder development cycle. This explains why our results in this batch are generally better on openstack-icps than on BonFIRE.

When simulating executions performed on openstack-icps 50% of our simulations end up with deviations lower than 7% on all metrics. On BonFIRE only 6% of simulations get this low an absolute error. The median absolute error on BonFIRE is systematically larger than the third quartile on openstack-icps. With the notable exception of the uptime on openstack-icsp, the third quartile error is over 20% and the worst simulations over 85% absolute error.

The *best* simulation batch. These simulations are performed with all of SimSchlouder additional capabilities. They represent the highest precision SimSchlouder can achieve.

We still observe higher precisions when working with openstack-icps than with BonFire. When looking at makespan, uptime, and usage, 98% of the simulations of executions performed on openstack-icps present levels of absolute error below 10%. Whereas with BonFIRE only 80% of simulations reach this level of accuracy.

The scheduling error metric presents both a higher number of simulations with levels of error over 10% but also higher values of errors on these simulations. The higher values are partially explained by the nature of the schederror metric. Scheduling decisions are

highly interdependent and a first scheduling error is likely to lead to several more as the scheduler plans around the first misplaced task.

Studying the simulations with high level of scheduling errors teaches us two things:

- Some of these first scheduling errors are due to choices made between equivalent placement solutions. In those cases slight differences in data structures, like list orders, can throw the simulation the wrong way.
- In other cases slight variations in timing, could alter the apparent availability of resources in the simulation.

Schlouder operates mostly on two threads, the node manager and the task manager, and the timing of context switch between the two affects scheduling, tasks' submissions and measured walltimes. Our `real_threads` flag only enables the simulation of precise scheduling dates. Information on the exact timing of other interruptions is not available. This creates a discrepancy since in SimSchlouder both threads can run simultaneously and instantly, as the simulation clock is suspended. This creates a situation where SimSchlouder always knows the true state of the world whereas Schlouder would often account for changes in a delayed fashion, leading to different scheduling decisions and therefore higher schederror rates. The much lower error rate in other metrics is a strong indication that most simulation scheduling errors will result in the simulation of equivalent scheduling solutions.

Providing `real walltimes`, `real boottimes`, and `real thread` an overall 90% of simulation, got below 10% absolute error on the exact makespan and uptime (cost) of the corresponding real execution. The three remaining scheduling batches study the impact of each of these flags showing how well the simulation performs without them.

- `real_walltimes` Simulations from the *no-walltimes* batch present results closer to the *blind* batch than the *best* one. Although providing the real boottimes and scheduling dates does reduce third quartile absolute error on makespan, uptime, and usage those gains do not significantly affect the first-quartile or median. Conversely, simulations where real walltimes are used will present lower values across all three quartiles. This makes it clear that correct walltimes are the biggest contributor to accurate simulations in terms of makespan, uptime, and usage. Scheduling error accuracy however does not appear to depend on the presence of real walltimes.
- `real_boottimes` The effect of real boottimes is best observed looking at the makespan metrics. Results in other metrics will be consistent with the results obtained in the *best* simulation due to the fact that these metrics are built in such a way that they ignore boottimes. The impact of real boottimes is experienced mostly through outliers. On both openstack-icps and BonFIRE boottimes have remained in most cases stable and consistent with the boottime estimators provided. However in some executions we observed widely inconsistent boottimes, with some BonFIRE executions presenting boottimes ranging from 5 minutes to 3 hours. The importance of real boottimes for accurate simulation depends mostly on the variability of real boottimes.

- `real_threads` The influence of injecting real scheduling dates through the use of the `real_threads` is mostly seen on scheduling errors. The repartition of scheduling errors in the *no-threads* batch is similar the one observed in the *blind* batch. Whereas *no-walltimes*, *no-boottimes*, and *best*, where real scheduling dates are available all share similar schederror repartitions. The `real_threads` metric also noticeably influences the other metrics but not on a scale of `real_walltimes`.

In conclusion, these simulations were conducted all along SimSchlouder's development as a way to measure the simulator's progress. Doing so helped both refine the simulator's precision and find bugs in Schlouder itself. This experiment shows that discrepancies between our base simulation and the observed reality can be attributed to three major sources:

- **Discrepancies in the user estimated walltimes.** The expected walltime provided to Schlouder and SimSchlouder for scheduling purposes will always deviate from the effective runtime of the task. This forms the brunt of the simulation error.

- **Discrepancies in the platform expected behavior.** In cases where the platform is stable in terms of boottimes the use of simple estimators should be enough to maintain simulation accuracy. In normal circumstances the impact of boottimes is limited by the fact that boottimes represent a significantly smaller portion of an execution than walltimes. However once a platform exhibits erratic behaviour, boottimes can start varying by orders of magnitude as we experienced with BonFIRE.

- **Discrepancies in the simulated behaviour.** SimSchlouder is capable of something Schlouder is not: instantaneous computations. By virtue of being written as SimGrid usercode, SimSchlouder executes itself as the simulation clock is suspended. This makes any action performed by SimSchlouder instantaneous in the world of the simulation. Schlouder however experiences small delays between events happening and the time it registers them. These discrepancies affect directly how SimSchlouder perceives the simulated state of the world. This leads to scheduling errors which aggravates errors from other sources of discrepancies. Our experience shows that even though scheduling dates certainly help in mitigating these kinds of discrepancies, most of the simulation error left in our best case simulation were caused by this divergence in the simulated behavior.

Our experiment shows that all these sources of discrepancies can be put in check using the appropriate available data. This is good news for SimSchlouder, since it allows us to prove its accuracy. It is however bad news for our plan of using SimSchlouder as a prediction tool. The necessary data requires the real world execution of a workload to happen before performing the simulation to provide data to the now pointless simulation. Moreover such a simulation is only valid for a single real execution. Even with all this additional data our simulation does not account for the fourth source of discrepancies: the inter-execution variability.

3.4 Take-away

To help users select the best scheduling heuristic for their needs we developed SimSchlouder, a simulation of Schlouder and the IaaS cloud it uses. SimSchlouder is based on SimGrid and SchIaaS. To insure that SimSchlouder faithfully reproduces Schlouder's behavior we set up a validation experiment comparing real execution traces to simulations reproducing those executions.

This experiment shows that accurate walltimes and boot times are key drivers of overall simulation accuracy. However exact reproduction of real executions, such as perfect reproductions of the schedule, requires precise simulation of the internal components of the simulated system. These components, such as thread switching, can only be measured by executing the workload. This hampers our attempts at using SimSchlouder as a prediction tool.

Part II

Stochastic Simulations

Chapter 4

Monte-Carlo Simulations

4.1 Motivation

In real executions performed for the evaluations of our simulator (Section 3.3) we observed a variability in task walltimes from one execution to the next. Using *relative standard deviation*, the ratio of the arithmetic standard deviation over the arithmetic mean, we were able to quantify this variability to be between 0.7% and 18%. When executions of a same workload on a same platform yield different results every time, how do we go about creating a prediction.

4.1.1 Sources of variability.

In scientific workloads. In instances where task inputs are strictly identical, such as our repeated executions of identical workloads, variability can come from two sources. First the tasks may use an element of randomness in their execution. The second source is background noise caused by externalities to the code. Those externalities take the form of interrupts, cache misses, page defaults, and I/O delays. This kind of variability is due to processes competing for access to resources, this phenomenon is called resource contention. Contention will depend on the number of processes needing a resource of a given type and the number of corresponding resources. Even on large systems where a single core can be dedicated to each process, other resources, such as L3 caches, are less numerous and will have to be shared between cores and therefore processes. I/O resources such as hard disk drives and network cards are not only resources with longer response times but also tend to be bottlenecks where contention often occurs.

This difference in resource availability can lead different applications to exhibit different levels of variability. Early in our work we studied opportunities to benchmark variability, however it soon became obvious that the differences in the resource usage profiles of our two applications, OMSSA and Montage, alone would make this endeavor complicated.

Contention has also been observed between the different tasks of a single workload.

Executions performed with AFAP exhibit shorter task walltimes than those with ASAP. This happens because in cases where the platform is not fully loaded ASAP will create more parallel processes than AFAP, therefore creating higher resource contention. The higher parallelism will allow ASAP to obtain shorter makespans in spite of the higher walltimes, however we have observed cases in which AFAP had shorter makespans outright. These kinds of observations motivate the need for prediction tools.

IaaS influences on variability. Virtualisation technologies have a performance overhead. Using Linpack ([31]) to benchmark the CPU we measured a 29% decrease in performance when performing the benchmark in a VM created using KVM-QEMU ([26]) instead of on the physical machine itself. As described in [20] this performance overhead comes from the masking by KVM of the underlying physical topology. However this performance reduction does not appear to come with higher variability. Our measures over multiple executions of Linpack show no significant variation in the relative standard deviation of the results obtained by the benchmark.

However if virtualisation does not appear to provoke higher variability, the economic model of IaaS contributes to higher variability. IaaS operators have an economic interest in mutualizing resources, therefore VMs are often grouped together on physical machines leading to higher level of contentions. In 2016 Leitner and Cito completed a survey[27] of different cloud platforms and cloud flavors. This survey gives a good view of the variability experienced on the cloud and the parameters that affect this variability. The survey was performed on three public cloud platforms, Amazon Web Services (AWS) EC2, Google Cloud Engine (GCE), and the Microsoft Azure cloud (Azure). The parameters studied include instance flavors, time of day, day of week, and application profile. In the survey Leitner and Cito conclude that whereas time of day and day of week had very little impact on performance and variability, the application resource usage profiles were very impactful on the measured variability. In their measures of relative standard deviation, Leitner and Cito found that CPU-bound applications had the most stable performance, with a relative standard deviation of 3.2% on EC2 m1.small instances in the Europe region, whereas applications generating high amounts of disk I/O presented the most unstable performance, with a relative standard deviation of 88.5% for the same instance type in the same region. In our own executions when comparing tasks' walltimes between comparable executions we observed an average relative standard deviation ranging from 0% to 30% averaging at 2.5% for OMSSA. For Montage tasks the average relative standard deviation was higher at 11%.

For the purpose of predictive simulations this variability is a major problem. The simulation performed in the previous chapter matched their real execution counterpart, as was the point of the experiment, but in doing so they failed at representing other executions of the same application with the same heuristic on the same cloud. In essence, even if we are able to perfectly predict one execution, it would still fail more often than it succeeds at predicting real executions simply from the fact that due to variability two same executions are unlikely to yield the same results. One could defend the result of such a simulation as "close enough" to the other executions. In practice however users are unwilling to trust a simulation that is wrong most of the time for any kind of decision making.

4.2 Stochastic simulations

To account for this inter-execution variability we shifted our focus towards stochastic simulations. In these simulations inputs are distributions of possibilities instead of predetermined values. Outputs also are distributions of possible results instead of single computed values. In our case the inputs will remain the tasks' individual walltimes, now expressed as distributions that account for the variability found in an IaaS context. The results would be distributions of makespan and costs. Although, distributions are more complex than single numerical values, they are able to represent multiple executions in a single simulation. Additionally, numerical tools from the field of statistics help quantify how often a real execution is expected to fall within the simulation result through the use of confidence intervals.

Our search on stochastic simulation lead us to two possible approaches. The first is a numerical approach for the resolution of stochastic static DAGs. The second approach uses Monte-Carlo methods based on the repetition of deterministic simulations.

4.2.1 Resolution of stochastic DAGs

Stochastic directed acyclic graphs (DAGs) is one way to represent scheduled workflows in variable environments. When considering our workload from a scheduling perspective, we can create a DAG where the vertices represent the tasks comprising the application, and the edges represent the scheduling dependencies between those tasks. The data dependencies can be omitted from the DAG provided the schedule already respects data dependencies. In a stochastic DAG, each vertex is assigned a distribution of possible runtimes. The simulation's expected result is a distribution representing the possible durations for the execution of the whole DAG, *i.e.* the *makespan*. Such a simulation would be a departure from the bottom-up approach used in SimSchlouder. We would consider our application only from the tasks' schedule perspective and the underlying infrastructure is not usually simulated. If we see the schedule generated by Schlouder as a form of dependency between tasks, then every workload is a workflow that can be represented as a DAG. The ability to abstract completely the underlying platform is useful when working with commercial IaaS clouds where the specifications of the underlying platform are usually unknown. This DAG based approach offers a good representation of an *offline* schedule, where the scheduling happens statically before the start of the execution. A number of works have studied the numerical resolution of such stochastic DAGs, in particular on heterogeneous grids [29] and on PERT networks [32].

For computation purposes distributions of possible outcomes, whether walltimes or makespans, are represented by random variables (RVs). An RV is defined by its probability density function (PDF) and cumulative distribution function (CDF), where the CDF is obtained by integration of the PDF. The numerical approach presented in [29, 32] shows that when task runtimes are independent, the makespan of successive tasks is a convolution product of the individual tasks' PDFs, while the makespan of parallel tasks joining is the product of the tasks' respective CDFs. Using these two methods makes it possible to fuse tasks within the stochastic DAG while computing the resulting task's

RV. Repeating the process as needed one can reduce the DAG to a single task whose RV represents the makespan of the entire stochastic DAG. However this approach is both computationally intensive and strongly reliant on the independence of the different task RVs.

Different methods exist to remove one or the other of these constraints. Dodin [17] proposes a method to reliably bound the results of stochastic *series-parallel graphs* and provides a mechanism by which any arbitrary DAG can be approximated in a *series-parallel graph*. This method is reliable even in cases where task walltime distributions are not independent. Spelde [50] proposes approximation based on the central limit theorem, in which task RVs are reduced to their mean and standard deviation. This approximation allows for the computation of a makespan RV without using any convolution allowing for a faster computation. However the independence of task RVs must still be respected with this method. The approximations required in both of these methods limit their use to bounding the DAG makespan distribution instead of computing it exactly.

All-in-all we found DAG-based numerical resolution poses two difficulties. First, it is impossible for us to guarantee independence between tasks, since the performance of the underlying VM can affect multiple tasks in a row. Outside of Dodin's approximation numerically solving a stochastic DAG with interdependent task distribution is deemed an intractable problem. Second, all of these DAG-based approaches imply a fixed schedule. Since the edges represent the scheduling dependencies and in our case Schlouder implements an online scheduling, the DAG might adopt a different structure depending on the runtime of certain tasks. This vertex to edge dependency also precludes the use of the aforementioned numerical resolution for our stochastic simulations.

In order to tackle these issues we consider in the following a different approach based on Monte Carlo simulations (MCSs). Used in 1963 by van Slyke to solve stochastic PERT graphs [48], Monte Carlo simulations were also used successfully to evaluate the robustness of offline schedules [11] in 2010. In 2017 *ElasticSim* [9] extended CloudSim with an MCS to integrate resource auto-scaling and stochastic task management. Similarly to our work, ElasticSim computes a schedule whose objective is to minimize rental cost while meeting deadline constraints. For several generated workflows, the study compares the simulation results regarding rental cost and makespan, when varying the variability of task duration and deadline with arbitrary values. By contrast, our work focuses on how the MCS method, under some given variability assumptions, captures actual observations.

4.2.2 Monte Carlo Simulations

Monte Carlo methods are a class of algorithms that rely on random sampling to obtain a result. Such methods can be used to solve numerical problems, like numerical integration, or stochastic problems. Although some variations exist depending on the nature of the problem, Monte Carlo methods usually work by drawing possible inputs from a given distribution, performing a deterministic computation on the inputs, and aggregating the results.

A simple example of such processes is the computation of the value of π via Monte

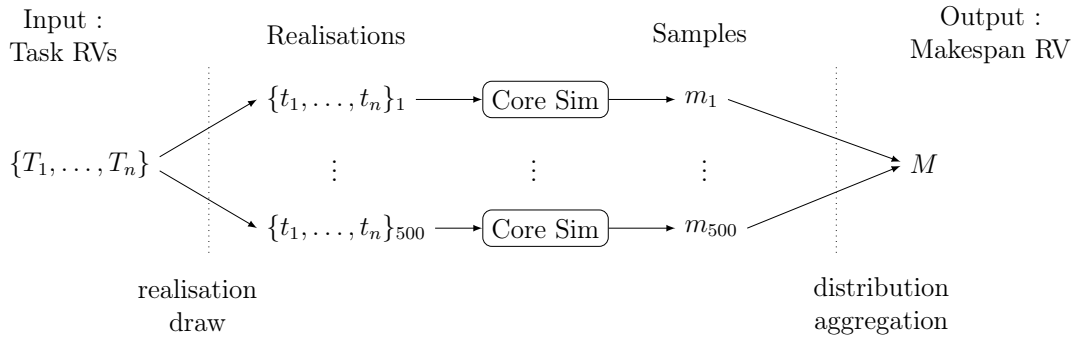

Figure 4.1: Overview of a 500-iteration Monte Carlo simulation.

Carlo integration. In this example a quarter circle of radius 1 is fitted in a square of size one. Random points within the square are drawn using an uniform distribution. Next each point's distance to the origin is computed to determine whether they fall within the circle boundary. Finally the results from every point are aggregated and the ratio of points falling within the circle computed. Given enough points this ratio will converge towards $\frac{\pi}{4}$. This example presents two points of interest of Monte Carlo methods. First, the input domain and the distribution must be properly defined. Second, Monte Carlo methods converge towards the wanted result as more samples are drawn.

Monte Carlo simulations (MCSs) are a subset of Monte Carlo methods concerned with the resolution of stochastic simulations. In Monte Carlo simulations the input domain is the full set of RVs used in the stochastic simulation. A realization is obtained by drawing a value from each of the RVs of the input domain. Each realization is a single point in a multidimensional input space of possible task walltimes. Since realizations only contain fixed numerical values they can then be simulated using a deterministic simulator, called *core simulator*. In our case we will rely on SimSchlouder. The results of every deterministic simulation provides a makespan and cost sample that can be aggregated to compute the MCSs result. An overview of the MCS process is shown in Figure 4.1 .

Monte Carlo simulations are advantageous because they allow us to build our simulations around deterministic simulators. Deterministic simulators are easier to build and easier to test than stochastic ones. The MCSs core simulator also allows us to include variables that could not be easily accounted for in stochastic DAGs. By using SimSchlouder as the core simulator of an MCS, all the online scheduling decisions that were not accounted for by the numerical resolution of stochastic DAGs are now simply solved at simulation time in a way that matches real executions.

4.2.3 Benefits of the stochastic approach

Using stochastic simulations is advantageous on three levels:

1. **Integrating inherent variability.** With task walltimes being replaced by a distribution of possible walltimes the simulation now directly acknowledges the variability observed in real life and accounts for these effects.

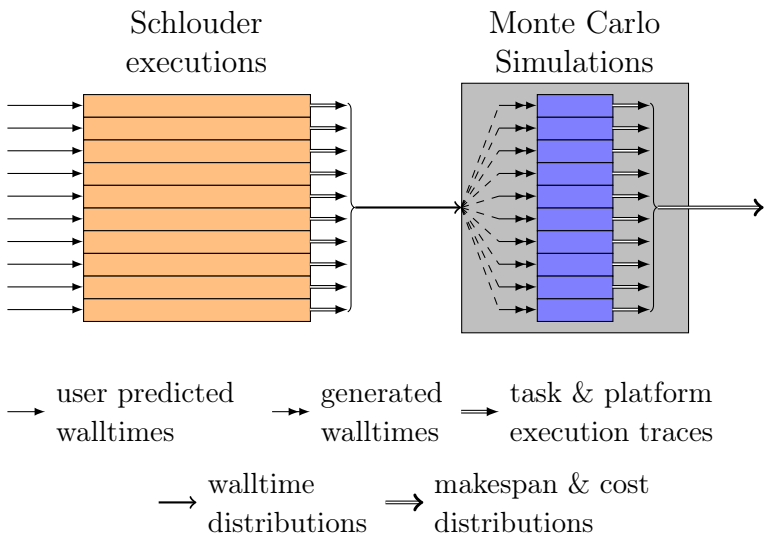

Figure 4.2: Representation of the MCS evaluation experiment. 1. Traces from real executions are used to produce walltime distributions. 2. The MCS's resulting makespan and cost distributions are compared to the real ones.

2. **Reducing dependency on precise timings.** We saw in Section 3.3 that the accuracy of SimSchlouder was visibly affected by the availability of timing information such as exact walltimes and scheduling dates. By providing walltime distributions instead of values, we do not need precise walltimes and scheduling dates. This alleviates the need to execute workloads to be able to time them and reduces the amount of information needed for precise simulations.

3. **Increasing trustworthiness in simulation results.** Stochastic simulations output makespan and cost distributions. This provides the users more visibility on the expected outcomes than a single deterministic simulation result.

Monte Carlo simulations allow us to easily build a stochastic simulator using SimSchlouder, which we have independently validated.

4.3 Our proposed Monte Carlo Simulation

In this section we build an MCS based on SimSchlouder. Such an MCS requires walltime distributions as inputs. As such we also propose an input model to provide such distributions. To find how our MCS and input model compare to reality we set up an evaluation experiment. In this section we will first present our experiment. Next we describe our reference real executions and our simulation setup. Then we will detail our input model and lastly we will present our experiment's results.

4.3.1 Experimental setup

We propose an experiment, represented in Figure 4.2, similar to the SimSchlouder validation experiment presented in Section 3.3.

The experiment takes an MCS and compares it to real executions. Since MCSs use distributions as inputs and as output, they can only be compared to a body of real executions instead of individual executions. Executions within a body are performed in similar circumstances, that is executions of a given workload on a given platform using a given scheduling heuristic. The validity of the MCS is tested by comparing the distribution of these groups of similar real executions to the distribution outputted by the MCS.

Our MCS also requires walltime distributions as input. These are built based on the walltimes observed in the real executions the MCS is trying to reproduce. Although the availability of the real executions allows us to build arbitrarily precise input distributions, our objective is still to experiment with the use of MCSs as a prediction tool. To reflect this, we will limit the amount of information required to build our input distribution. The chosen distributions, called input model, and the reasoning behind them are described in Section 4.3.4.

By quantifying how well our MCS captures the distribution of real executions, we evaluate the usability of MCS as a prediction tool under our self-imposed input model constraints.

4.3.2 Real executions

The major difficulty posed by this experiment, for the real execution side of the experiment, is the fragmentation of our trace log archive. As described in Section 3.3, we had 336 execution traces mostly issued from the development process of Schlouder. Although this represents a significant number of traces by itself, the validation experiment we are building here requires groups of executions performed under similar circumstances. On this point the traces collected previously proved inadequate. During development, Schlouder and the IaaS platform on which the executions were performed were liable to change regularly and executions were pretty uniformly spread between applications and use-cases. Therefore within the archive most groups of equivalent runs are rare, and seldom contain more than 2 or 3 equivalent runs.

To provide a suitable reference for this validation we performed specific executions of OMSSA. The reference executions were performed using OMSSA on the BRS use-case. As detailed in Section 1.3.1 the BRS use-case of OMSSA contains 233 tasks and is overall the biggest OMSSA use-case available to our team. This new set of executions contains 106 runs performed with the ASAP scheduling heuristic, and 100 runs with the AFAP heuristic.

The executions were performed on our local cloud testbed. In these executions the cloud was configured to use two compute nodes. Compute nodes are built with two 2.67 GHz Intel Xeon X5650 each with 6 hyper-threaded cores for a total of 24 cores per node. Nodes are operated on Ubuntu 2014.04, and used the cloud system Openstack

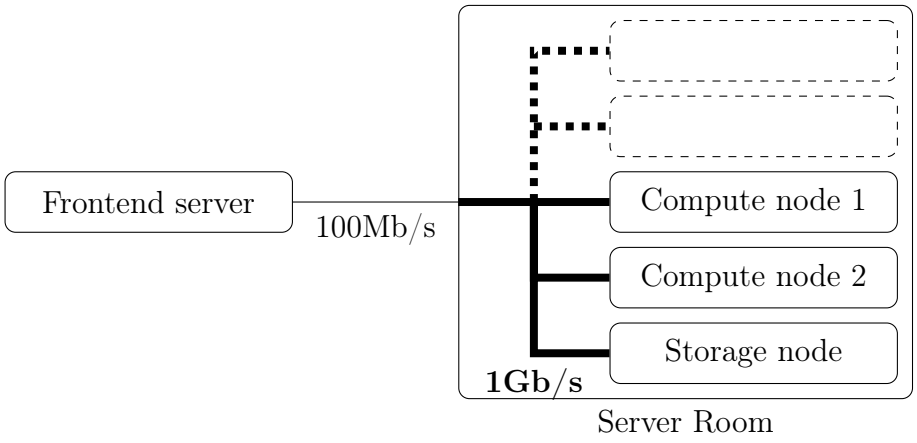

Figure 4.3: Network infrastructure of the private cloud testbed used in real executions. *The Frontend server runs Openstack administrative functions and the Schlouder server. The Compute nodes operate the VMs, and the Storage node stores the tasks' inputs and outputs.*

2014.4 with KVM/QEMU as a hypervisor. During these executions the cloud was configured to limit the number of simultaneous VMs to 10, limiting to some extent the level of contention we will observe. However the network on which the compute nodes communicate is not isolated and is part of the University's shared infrastructure. Moreover the machine hosting the cloud front-end and the Schlouder server are on different networks with limited bandwidth as shown in figure 4.3.

Figure 4.4 presents the distribution of makespans and costs for the real executions described above. Since we don't actually pay to use the testbed the cost is given as BTU count, the number of open BTUs during the experiment, with a BTU length arbitrarily set to 3600s (1 hour). In these real executions we observed:

- For ASAP: makespan ranges [12811s;13488s] (variation of ≈5%) with a constant BTU count of 40
- For AFAP: makespan ranges [13564s;14172s] (variation of 4%) with a BTU count ranging [33;36].

When looking at variability on the task level, using the same relative standard deviation method as used in [27], we observed on average a 4% variability with the median task variability being 2%. Overall only 10 tasks showed a relative *sd* exceeding 3%, all of those seemed to be linked to a single execution[1]. Since this experiment is about measuring the effectiveness of MCS in capturing execution in the face of variability, this execution was not removed from the data set. This variability is within the range reported in the survey [27] for platforms like Amazon's EC2 or Google Cloud Engine.

[1] namely execution `v4.montecarlo-brs-10.brs.afap.regular1gb.openstack-icps.v5.92`

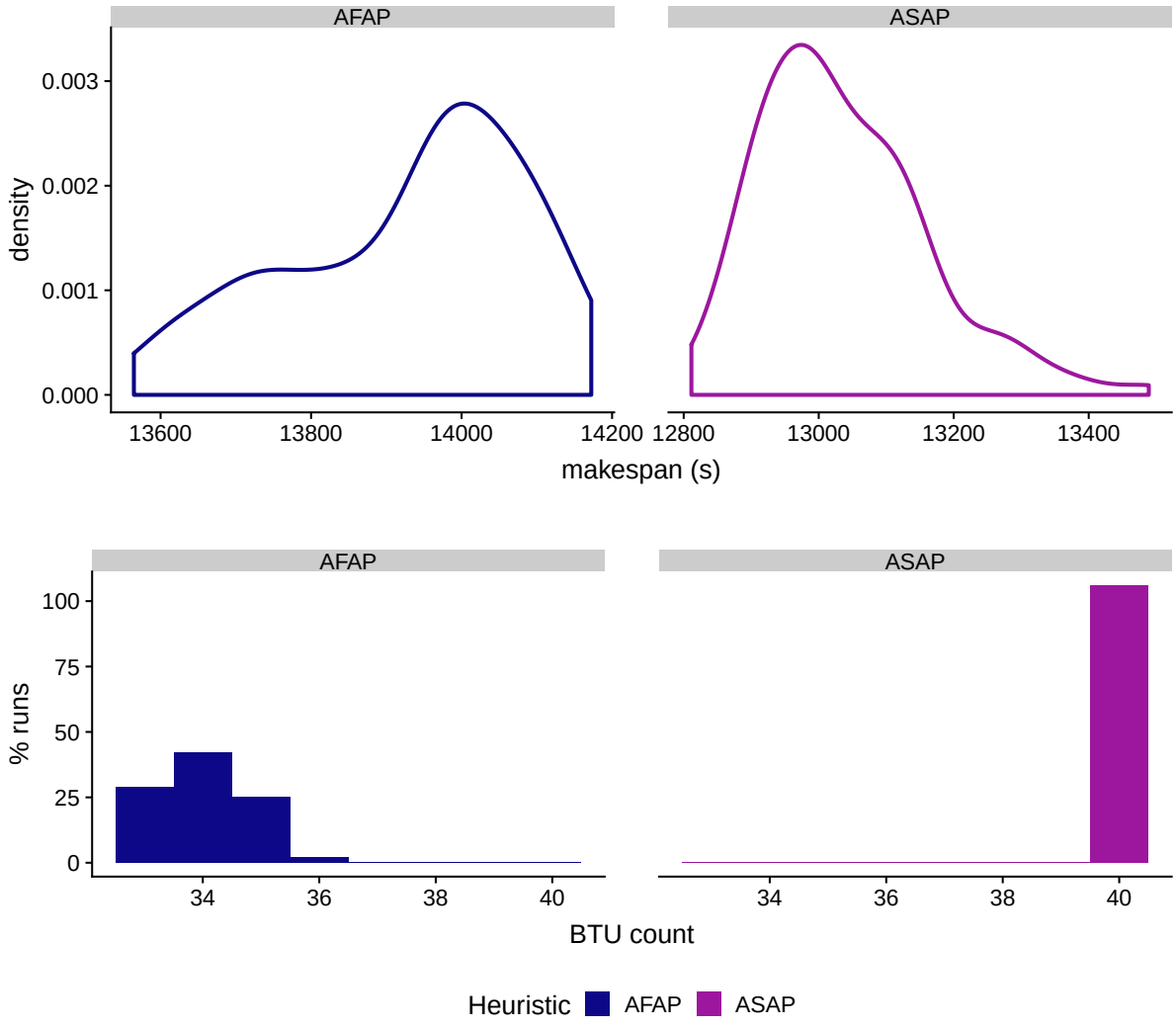

Figure 4.4: Empirical observations for makespan and BTU count distributions.
Reading example: Using ASAP leads to makespans roughly ranging from 12800 s to 13600 s and BTU counts of 40.

4.3.3 Monte Carlo Simulation tooling

We implement our Monte Carlo simulation using SimSchlouder as our core simulator. This is convenient since we know from Section 3.3 that SimSchlouder is relatively precise and capable of reproducing the scheduling produced by Schlouder. Instead of implementing our MCS as a new application using the code from SimSchlouder, we decided to repurpose the `lab.py` script presented in 3.3.

Configuration 4.1 `montecarlo-cmp.cfg`, the lab configuration file used to run a Monte Carlo simulation using SimSchlouder as the core simulator.

```
SETUP_DIR ./setup/simschlouder
NEEDED xml/*
PRE_COMMAND_SETUP ./montecarlo-cmp-pre.sh reference.tasks > mcsim.cfg
POST_COMMAND_SETUP ./montecarlo-cmp-post.sh > montecarlo-cmp.metrics.dat

SIM_ARG 1 simschlouder.SimSchlouder
INCLUDE mcsim.cfg
SIM_ARG 3 real_walltimes
```

The configuration used to execute an MCS using `lab.py` is included as Configuration 4.1. As with the SimSchlouder validation process a huge section of the configuration is generated at runtime by the pre-command script and then included in the middle of the SIM_ARGs.

The pre-command script is tasked with generating the realisations used in the MCS, it works by taking a template task file as argument and a few parameters set in the script itself to generate hundreds of different task files each of them with different task runtimes drawn following the input model we will detail in section 4.3.4. Every time a new task file is generated the pre-command script writes a corresponding SIM_ARG line to the standard input which is then redirected to the included file. The generated SIM_ARG lines indicate 2 as index number and provides to SimSchlouder the correct SimSchlouder configuration file, the generated task file, and the correct scheduling heuristic for this task file.

The generated task files contain task names, submission dates, and expected walltimes copied from the template task file. These are the same values as those used during the real executions, and are used across all realisations. Additionally the generated task files contain real walltimes drawn by the pre-command script. These are drawn following the input distributions and differentiate the different realisations from one another. The `real_walltimes` flag insures SimSchlouder uses these drawn walltimes during the simulation time.

Once every realisation has been simulated by SimSchlouder, the post-command script is charged with the aggregation phase presented at the end of MCSs, as shown in figure 4.1. Ideally this script should be able to take the outputs of all the SimSchlouder executions performed during the MCS and compute distributions of possible makespans and costs before outputting them in a legible format. However, we only automated the aggregation of SimSchlouder results in a single file, the analysis of said results and the eventual fitting

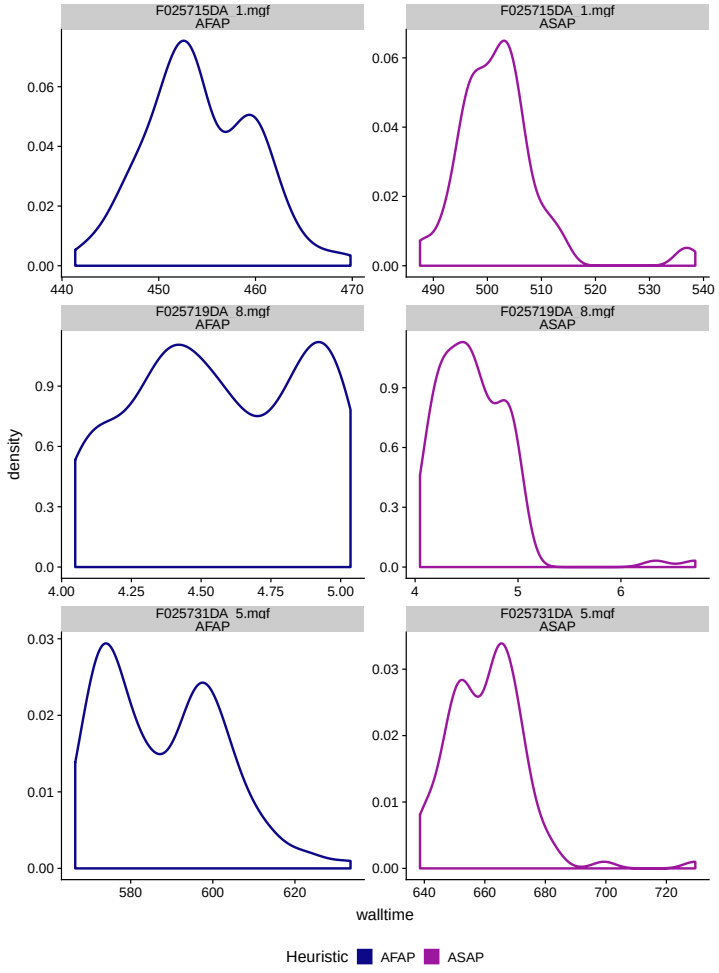

Figure 4.5: Density distributions for representative tasks taken from our reference executions.

were performed with R after the execution of the MCS was finished.

4.3.4 Input modeling

Observed Distributions

Figure 4.5 presents the walltime density distributions for a few tasks in our reference OMSSA run. These show the diversity of walltime distributions observed in real executions. Distributions appear to vary from task to task and depending on the scheduling heuristic. This is what the input distributions should try to represent. Choosing distributions to which to fit a given data set is a complicated matter.

- Task `F025715DA_1.mgf`(top row) displays a noticeable shift in walltimes depending on the scheduling heuristics. The walltimes when using the AFAP range from 440s to 470s whereas when using ASAP the walltimes range from 480s to 540s. When

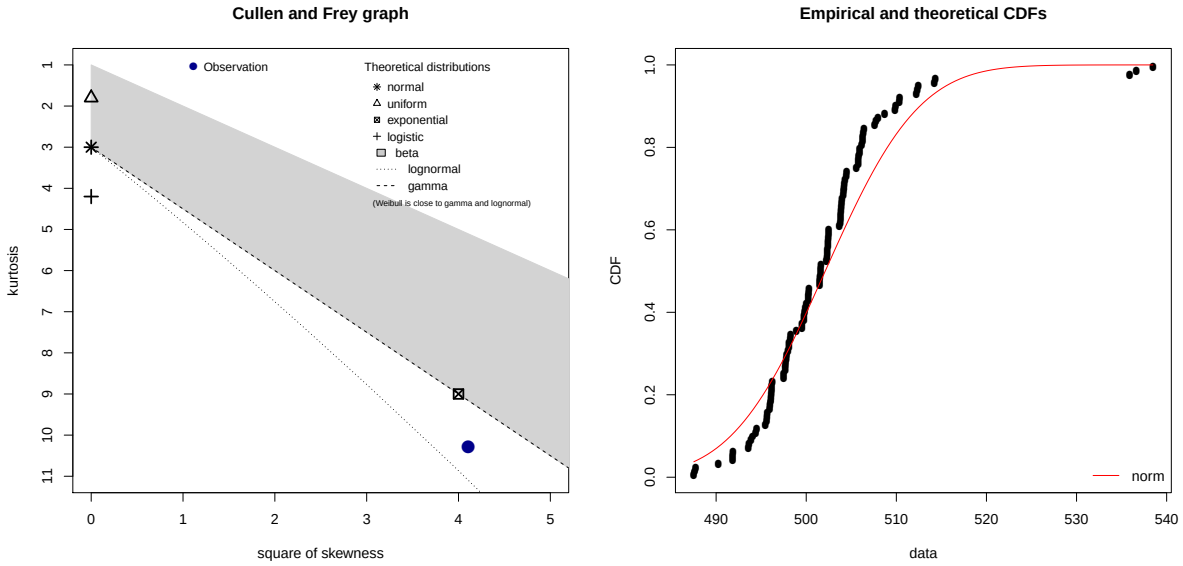

Figure 4.6: Cullen and Frey graph and empricical cumulative distribution function for the observations of tasks F025715DA_1.mgf with ASAP scheduling.

trying to fit a distribution to `F025715DA_1.mgf`, we are tempted to consider the distribution mostly normal.

- Task `F025719DA_8.mgf`(middle row) is noticeably shorter than the other two. This is due to being a partial spectra task. This task also does not display a significant shift based on the heuristic. For this task the AFAP distribution is bimodal and the ASAP one unimodal but it shares the asymmetric tail observed in `F025715DA_1.mgf`.

- Task `F025731DA_5.mgf`(bottom row) also displays a shift in walltime values between AFAP and ASAP. However unlike `F025715DA_1.mgf` this task's walltime distributions appear to be bimodal. In ASAP we observe the same long tail of higher outliers as before.

Looking more closely into the possible normality of the walltimes distribution of task `F025715DA_1.mgf` we applied two statistical tools to the distribution obtained when using the ASAP scheduling. These are shown in Figure 4.6:

- the Cullen and Frey graph which compares the arithmetic skewness and kurtosis of the empirical data to those of well-known distributions. The blue dot represents the skewness^2/kurtosis position for the observed data. These types of graphs are meant to help users find a distribution fitting their data.

- the empirical and theoretical CDF comparison graph for a normal distribution. This graph plots on the same plane the empirical CDF of the observed walltimes and the CDF of the closest normal fit. These types of graphs are usually used to evaluate the quality of a fit after the fitting has been done.

These show that even task `F025715DA_1.mgf` when using ASAP scheduling does not in fact match a normal distribution. All indicates that the 3 executions with walltimes over 535s are too significant to be considered as simple outliers as we thought would be possible.

This highlights the main difficulty of creating an input model. Fitting is a complicated process for non-statisticians like us, and most of the distributions presented here do not appear to match any theoretical distributions we are familiar with. Moreover the different tasks of OMSSA run the exact same code, the only difference between tasks being the input file, after which the task is named. Therefore the different distributions obtained here are all the results of the same code. We are also working with hundreds of empirical observations, which is much more than could be expected in a real life scenario. Making precise distributions from these observations would put us in the same situation we had with SimSchlouder, where precise simulation could only be achieved after real executions. MCSs made in a prediction context will really be limited to simple input distributions. This is what we intend to evaluate in this chapter.

Model choice

In batch scheduling users are often asked to provide an expected walltime for every task. Asking the users to provide estimates is an integral part of Schlouder operation. In most cases the users acquire these estimates either through advance knowledge of the workload's inner workings, short scale executions on local consumer hardware, or experience of previous executions of the workload. In any case users will intuit the value based on a fairly low number of sample points. This strongly limits what users can be expected to know. We contend that users possessing enough information to estimate an expected walltime could also estimate a minimum or maximum expected runtime. We propose an input model that makes use of this already available walltime and the presumptive spread estimate.

This fairly limited level of information means that mapping the difference in distributions observed in Figure 4.5 is impossible. We intend to show that this limitation does not preclude accurate MCSs. Our model takes the already provided expected walltimes and places them at the center of a uniform distribution. We do not expect user to form an opinion on the variability of each task individually, especially in cases like OMSSA where every task runs the exact same code on different inputs. Because of this the width of the uniform distributions will be fixed relatively to estimated runtime via a single parameter used for the whole workload. This new parameter is given as a percentage of the expected walltime and is call *perturbation level* (P). As such for a given task with an expected walltime r_e the input distribution will be:

$$\mathcal{U}(r_e \times (1 - P), r_e \times (1 + P)) \tag{4.1}$$

This approach involves only one parameter, the perturbation level, added to those already necessary for Schlouder or SimSchlouder operation. It also separates to some extent information about the workload from information about variability. In cases where the simulation is provided by the platform/scheduler operator, they could provide the perturbation

level based on their long term knowledge of the platform and its stability. Obviously such a simplistic model does not capture all the complexities of a real execution. Limiting ourselves to uniform distributions and binding all the distributions to a single perturbation level are significant approximations and some information loss is expected. The results of this experiment will allow us to determine if sufficient information remains to predict the workflow behavior.

Experimental values

To validate this model we must perform our MCS with the best parameters in our input model. We intend to limit errors due to misestimated walltimes or a misestimated perturbation level. Doing so we will guarantee that any error remaining in the MCS's results are due to our chosen input model. This is akin to our best-case simulation in SimSchlouder's validation experiment where we provide exact walltimes, boottimes, and thread times.

To provide the most accurate possible expected walltimes we opt to use the average observed walltimes during the real executions. These expected walltimes will end up in the center of the uniform distributions and therefore should become the mean observed value in the simulations. If we denote $\bar{r_j}$ the average walltime observed in real executions of a task j, the walltime distribution's RV T_j for the same task j is:

$$T_j = \mathcal{U}[\bar{r_j} \times (1 - P), \bar{r_j} \times (1 + P)] \tag{4.2}$$

Providing an accurate perturbation level however is more complicated. In our model P represents variability by establishing the limit for the worst deviations possible from the expected walltimes. The only metric of variability we have used in this work until now is the relative standard deviation used when discussing the findings in [27]. Metrics based on standard deviation relate more closely to the normal distributions which have asymptotic edges and are therefore not appropriate to use with the uniform distribution which has finite bounds. We need to find a way to define P that fairly represents the deviation for the means observed in our real executions, without giving undue weight to extreme outliers. We choose to build P by measuring for every task the worst observed deviation from the average walltime, and averaging these worst deviations across all n tasks. With r_j^n the walltime observed for task j in the nth real execution of the workload, P is set to:

$$P = \underset{j}{\text{mean}} \left(\max_n \left(\frac{|r_j^n - \bar{r_j}|}{\bar{r_j}} \right) \right) \tag{4.3}$$

In our real executions, the perturbation level given by this model is $P \approx 10\%$ for both heuristics. Using a similar metric Kim et al.([23]) also observed most deviations to be within 10% of the average walltime when working on Amazon EC2 instances with dedicated CPUs. This method choice to compute P is problematic because there is no clear cut, good estimator like with the average walltime. This makes the choice of P a potential source of error separate from the model itself. In the next section we will see

how this method of computing P yields perfectly acceptable results. The influence of P on the accuracy on the simulations results will be discussed in Section 5.1.

4.3.5 Results

Observed results

We executed 500-iteration MCSs for both heuristics using the task model described in the previous section. Additionally we also executed MCSs for the Montage workload based on the most recent series of executions from our archived executions. The resulting makespan and BTU count distributions are shown in Figure 4.7. The makespan density graphs show the simulation result distributions as filled curves. The real observed executions, as in Figure 4.4, are shown as non-filled curves. On the BTU count graphs, the left bar represents the empirical data, and the right bar the results from the simulation.

Looking at the makespans we can see the distributions outputted by the MCS fall globally within the same range as as the distributions of the real executions. However, the simulated distributions do not share the same shape as their real life counterparts. This is mostly visible when comparing the real distribution to the MCS's one in the OMSSA AFAP scenario. The real distribution presents a small plateau around $13800s$ which is absent from the simulated distribution. This results in both distributions presenting different modes, with the real distribution reaching its peak at a higher value than the simulated distribution. Globally the makespan distribution appears more symmetrical than its real counterpart. This is due to the approximations done in the input model. By reducing every task to uniform distributions we effectively erased all the subtleties that make the asymmetries present in real distribution possible.

The effect is much less striking when looking at the BTU count metric due to it being a discrete variable. Once again we see in the case of OMSSA AFAP that the MCS obtained the same values as those observed in real life, but the probability credited to each possible outcome diverges from those observed in real life, with the simulations showing a lower probability of costing 34 BTU than in real life.

Quantitative analysis

As discussed earlier the distributions outputted by the simulation are mostly symmetrical and bell-curved shaped. We reduced input distributions to a simple uniform distribution, which has a finite variance, and the distributions are independent, from a probability point of view. And because the makespan is the sum of the walltimes of the tasks along the critical path, we consider the Central Limit theorem applicable. This means that we would expect the simulation results to tend toward a normal distribution.

Using statistical fitting we can estimate the values of μ and σ for the normal distribution that match the simulation results the most closely. Those can be used to build confidence intervals (CIs). Confidence intervals indicate a range of possible values a random sample from a given distribution is expected to take. A CI has an associated

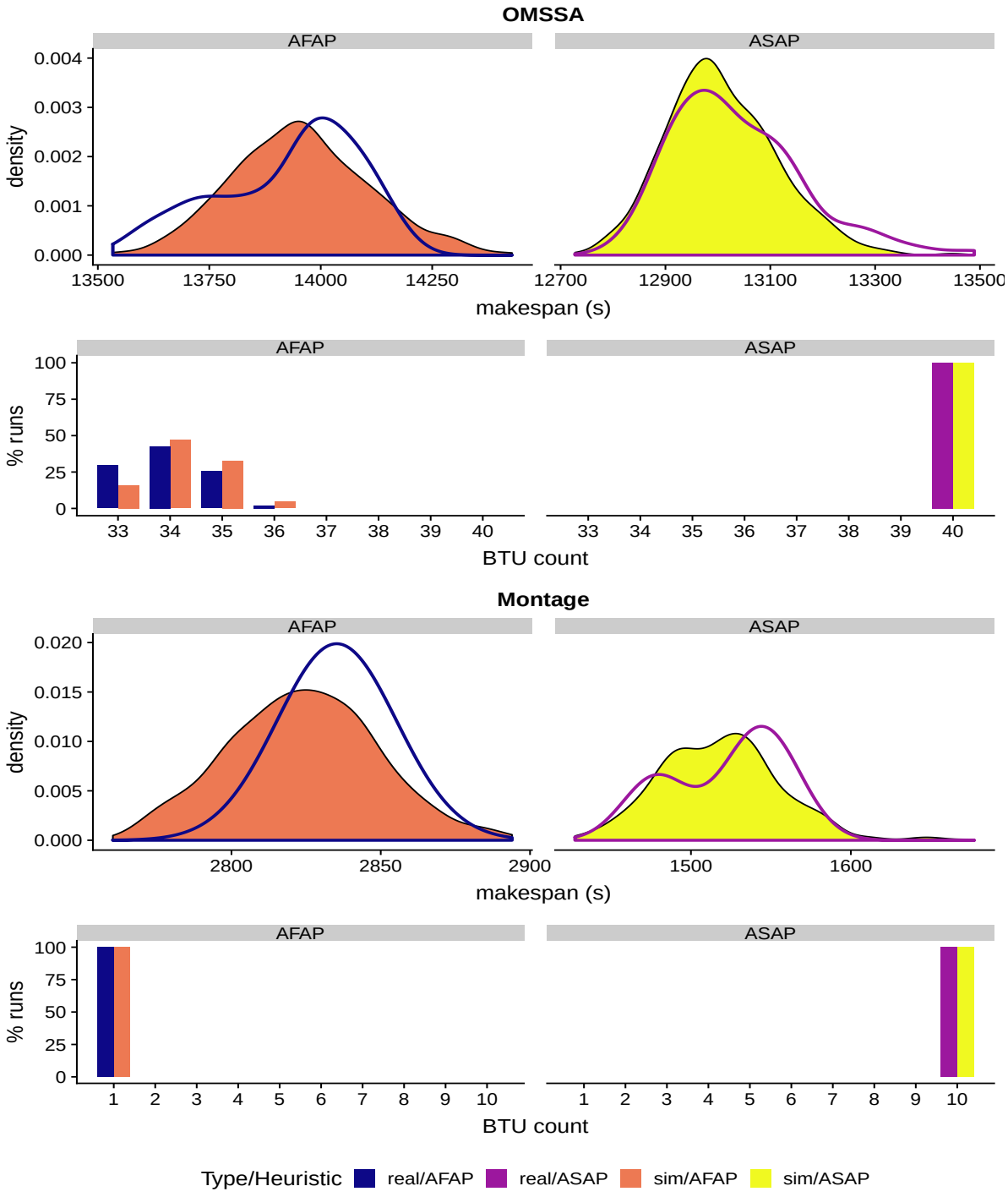

Figure 4.7: Makespan and BTU count distributions for MCS compared to reality for $P = 10\%$.
Reading example: Comparing the makespan distribution for OMSSA with AFAP to the real one(top left), we can see the simulation underestimates the likelihood of executions taking between 13500s and 13750s and between 14000s and 14200s. Looking at BTU counts (second row), we can see the simulation underestimates the likelihood of an execution taking only 33 BTUs.

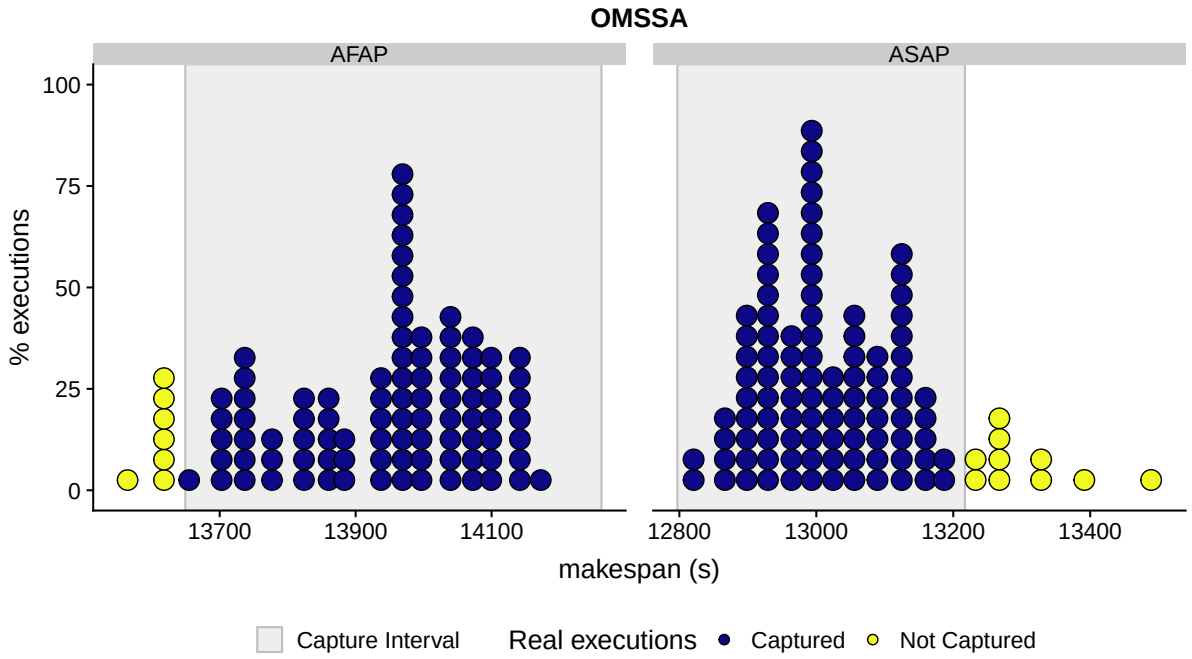

Figure 4.8: Demonstration of the capture rate on the OMSSA results for a 95 %CI. *Reading example: when using the ASAP heuristic, the 95% CI for the simulation results distribution ranges* $[12797.1, 13216.97]$. *This range contains 96 of our real executions (in blue) out of 106 total real executions (blue+yellow points) resulting in a 90.6% capture rate.*

confidence level that describes the ratio that a random sample from the distribution falls within the interval.

For the normal distribution, the 95% CI, that is a CI with a *confidence level* of 95%, is defined as $[\mu - 2\sigma, \mu + 2\sigma]$. This interval should contain 95.5% of the samples taken from a normally distributed population using the same μ and σ. Additionally the 99.7% CI ranges $[\mu - 3\sigma, \mu + 3\sigma]$. As with most statistics these confidence intervals tend to only be true with sufficiently high number of samples.

When fitting the results of our MCS to a normal distribution, we can produce the corresponding CIs. In order to quantify how well the MCS models the observations, we define a metric called *capture rate (CR)*. Given n real execution observations and the subset of the m observations whose makespan fall within a given CI, CR is defined as the ratio $\frac{m}{n}$. This process is illustrated in Figure 4.8. If the simulated distribution matched up with the real distribution the empirical CR should tend towards the theoretical confidence level of the CI being used as capture interval.

Table 4.1 presents the capture rates obtained by each interval computed after normal fitting. Additionally we provide for each interval its size relative to the average makespan.

Regarding OMSSA, the MCS captures at least 90% of real observed makespans. The divergence between the CR and the CI confidence level is due to multiple factors such as : the fact that the empirical makespan distribution is not following a perfect normal

Application	Heuristic	Makespan (Size of CI) CI 95%	CI 99%	BTU
OMSSA	ASAP	90% (3%)	98% (5%)	100%
	AFAP	92% (4%)	100% (6%)	100%
Montage	ASAP	100% (2%)	100% (4%)	100%
	AFAP	100% (1%)	100% (2%)	100%

Table 4.1: Makespan and BTU capture rates depending on CI for P=10%.

distribution, the multiple approximations done in the input model, and the low number of real executions from a statistical point of view. Using a 99% CI improves the capture rate up to 98%, very close to the theoretical expectation. Regarding Montage the MCS achieves a CR of 100% for both CIs.

As expected the capture rate falls short of the confidence level. This is not surprising since even if the simulation approaches a normal distribution, we saw in figure 4.7 that the real execution did not share such a distribution. Despite the approximation imposed by limiting ourselves to a simple task model, our MCS still managed to capture 90% of reality all the while producing makespan intervals of limited size, a 3% relative size representing 7 minutes on a 3h 45m long makespan. We consider this result a satisfactory trade-off between the simplicity of the input model and the accuracy with regards to the theoretical CI. In the next chapter we will discuss the effect of the perturbation level on the CR, the number of necessary simulations within the MCS, and other limits to the approach we have encountered.

4.4 Take-away

To build and enhance our simulations so as to enable us to use them as a prediction tool we shifted our focus toward stochastic simulations. By integrating variabilities for which SimSchlouder could not account for, stochastic simulations allow us to provide more precise predictions, without requiring as much information.

Monte Carlo simulations (MCSs) allow us to perform a stochastic simulation by repeatedly executing instances of SimSchlouder. This allows us to benefit from SimSchlouder's accurate reproduction of Schlouder's heuristic while sidestepping the computational intractability of the numerical resolution of stochastic dynamic schedules.

To keep with the prediction tool use-case we propose an input model which captures the entirety of the systems variability in a single perturbation level P. This variable is used to create uniform distributions around every task's expected walltime.

Through experimentation we show that MCSs using this input model can accurately represent 90% of real executions.

Chapter 5

Defining Simulation Inputs

In the previous chapter we establish our motivation for using stochastic simulations. We chose to do so using Monte Carlo simulations, and establish a model for task walltimes, and finally tested our stochastic simulations against real traces. In this chapter we study different aspects to show possible improvements on the model and the method. First we show the impact of the selection of the pertubation level P on the simulation results. Then we will discuss how the number of realisations in the MCS affect the results and their stability. Last we will discuss results obtained using a normal distribution instead of a uniform one in the input model.

5.1 P: the perturbation level

In Section 4.3.4 we raised the possibility that our choice for the perturbation level value, might be suboptimal for our stated objective of building the best simulation possible with our chosen input model. A subsequent question is whether the perturbation level can be used as a trade-off variable, between capture rate and output interval size. We hypothesize that as the perturbation level gets higher, the output distribution becomes wider, resulting in higher capture rates when building confidence intervals. This would mean that, assuming the relation between capture rate (CR) and perturbation level could be quantified, a user could adjust the simulations' CR to fit their needs, at the cost of larger intervals.

To test this hypothesis we performed Monte Carlo simulations with varying perturbation levels. Using the method presented in Section 4.3.5 we build a capture interval to evaluate each simulation's capture rate. Figure 5.1 shows the evolution of the capture rate as the perturbation gets higher for OMSSA. The behavior observed in the AFAP simulations matches our prediction. As the perturbation level increases, so does the width of the confidence intervals obtained by the MCS, resulting in higher capture rates. However, the behavior observed in MCS using the ASAP heuristic, diverges strongly from our early expectations. Although capture rate initially increases, this does not persist for higher values of P. At $P = 40\%$ the simulation's capture rate falls below the capture rate obtained at $P = 10$.

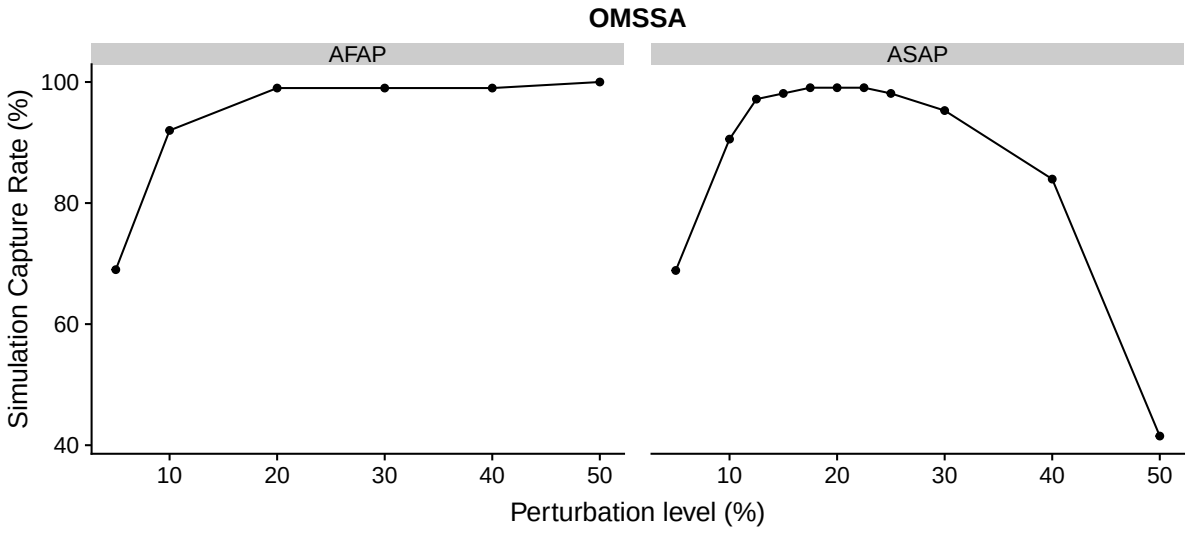

Figure 5.1: Capture rates obtained by simulations using different values of perturbation levels.
Reading example: When simulating OMSSA with the ASAP scheduling heuristic and a perturbation level of $P = 30\%$ *the capture rate of the simulation is* 95%.

Figure 5.2 presents the density distributions for real executions as well as the MCS distributions obtained at $P = 10\%$, $P = 20\%$ and $P = 40\%$. These distributions are represented using violins plots where the width of the shape represents the probability density. The point at the center of each shape provides the value of the mean observed makespan whereas the bar crossing the shape indicates the position of the median makespan.

As we expected the simulations performed with higher perturbation levels do produce wider distributions and therefore larger confidence intervals. However these simulations generate distributions with higher means. In the case of OMSSA with the ASAP heuristic this change in the obtained mean is significant compared to the growth of the distribution's width. This results in an offset capture interval with lower capture rates. The CR of the simulation with $P = 40\%$ is of only 83% when the $P = 10\%$ simulation had a 90% CR. OMSSA with ASAP is our only test case to present such a degradation of simulation performance. With other test cases using $P = 40\%$ resulted in equal or higher CRs. Both OMSSA's nature as a *bag-of-tasks* and the ASAP heuristic contribute to the drift phenomenon:

- OMSSA being a *bag-of-tasks* means that all scheduling happens at the start of the execution. Indeed Schlouder's online scheduling happens as soon as all task dependencies are cleared, and bag-of-tasks do not have any task dependencies. Because of this design Schlouder has no opportunity to correct scheduling once a task exceeds its expected walltime.

- ASAP by scheduling new jobs to the less charged VMs will tend to produce schedules where every VM is equally full. In such a context every task can affect the makespan by exceeding its expected walltime. The slowest task is more likely to be in the critical path by virtue of being the slowest task. When using AFAP the scheduling

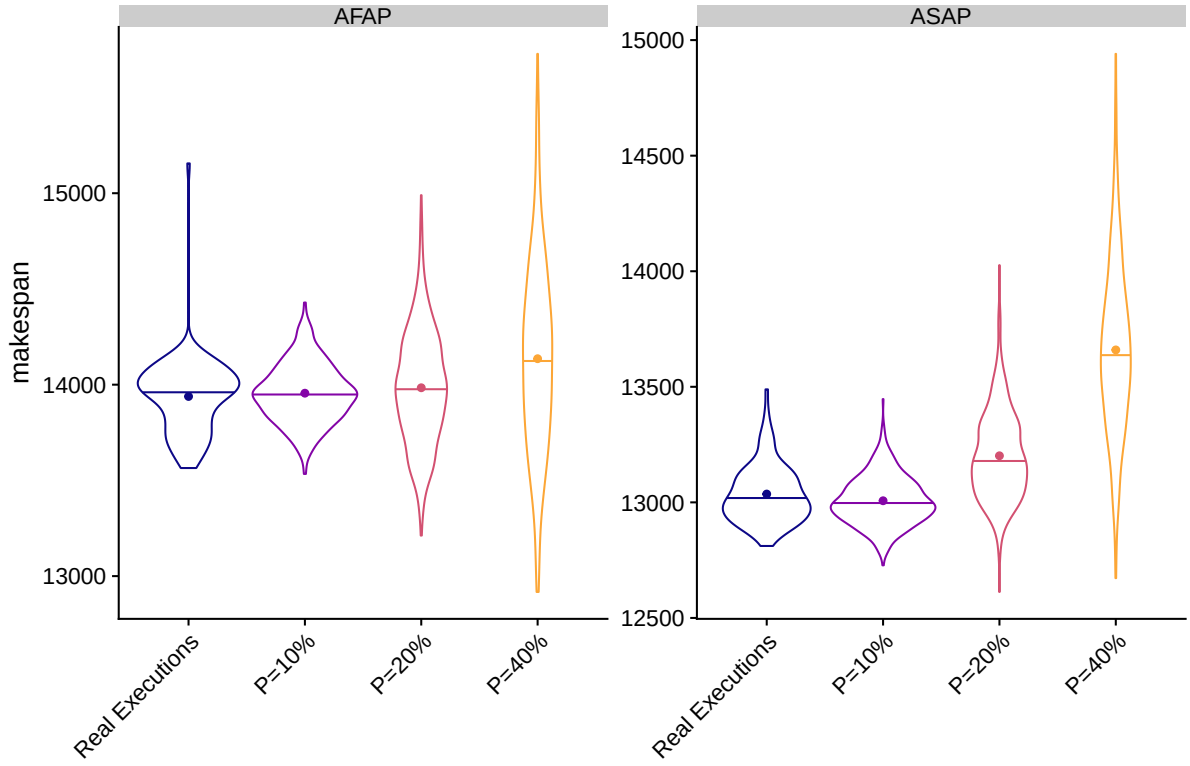

Figure 5.2: Makespan distribution obtained for OMSSA at different perturbation levels. The width of each shape is the probability density. The point indicates the average value. The bar represent the median observation.
Reading example: Looking at real executions performed in AFAP (leftmost column in the left graph), we observe the density peak at a makespan of 14007.26s, the median makespan is at 13973.76s, and the mean makespan value is 13938.37s.

will result in some VMs being more full than some others. As such only the tasks on these VMs can impact the makespan, slow tasks on other VMs are unlikely to result in longer makespans.

This shows the potential of Monte Carlo simulations for offline studies of application behavior in the face of variability. By varying P we were able to expose ASAP's weakness in variable environments. Although our stated objective is to build a prediction tool, MCSs are also an interesting scientific instrument to study the behavior of IaaS environments and the applications using such an environment. MCSs were already used in this fashion by Canon and Jeannot in [11] to study the robustness of DAG scheduling on grids presenting high variability.

Our experiment shows that the perturbation level should not be used as a trade-off variable to augment the capture rate at the expense of CI compactness. Users willing to trade off interval compactness for higher capture rates should use statistical methods to build higher rate CIs, like the 99% normal distribution CI used in Section 4.3.5. This also implies that choosing the correct value of P is critical for correct simulation.

Finding the best possible value of P increased the number of simulations for ASAP in figure 5.1 using a binary search approach. We observe our highest CR (99.05%) for the values of P of 17.5%, 20%, and 22.5%. A limiting factor in this approach is the discretization of the CR due to the limited number of real executions against which the CR is tested. Using $P = 20\%$ also results in a 99% CR for the MCS of OMSSA using the AFAP heuristic, although this result is also true of $P = 40\%$ as mentioned previously.

Although this approach for finding a value of P seems promising it is not without caveats:

- Firstly this approach relies on capture rate computation. Since the CR is computed based on real executions this approach does not fit the prediction tool constraint we have set for ourselves. However in environments where variability is mostly due to resource contention, and in which cross-traffic is relatively stable, users repeatedly using this platform might be able to collect enough real execution traces to perform this kind of optimisation of the value of P.

- Secondly, P is being used in the context of the proposed input model. Our input model uses P to size a uniform distribution around the task's expected walltimes. However we saw in Section 4.3.4 that most tasks do not conform to a uniform distribution in reality.

- Thirdly, our confidence interval are only rated for 95%. The CRs tested here are computed using the 95%CI obtained from the MCS's result. In this context the statistical significance of any CRs over 95% is limited. There is strong possibility that any CR above the 95% threshold might vary randomly from MCS to MCS. If we accept the CI's nominal value strictly then our results indicate that any value of P between 12.5% and 30% is acceptable, greatly limiting the usefulness of this approach.

- Lastly, MCSs are random processes. The random nature of the MCS means that results can vary from one execution to another. This would lead to different versions of Figure 5.1 and different values of P maximizing the CR. However we will see in Section 5.2 that we can limit inter-MCS variability by using a higher number of iterations.

Even accounting these reservations, this experiment shows that the approach used in Section 4.3.4 produced sub-optimal results. And for users possessing the required amounts data, and the time necessary for multiple rounds of simulations, this approach provides an opportunity to refine the value of P used in their simulations.

5.2 N: the number of iterations

All the MCSs presented up to this point used 500 iterations ($N = 500$), meaning our MCSs were comprised of 500 deterministic simulations based on inputs drawn randomly from our input model. Such MCSs require on average 15 minutes of CPU time. Like most statistical and stochastic processes, MCSs tend towards a result given a high enough number of iterations. In this section we explore this phenomenon from two angles. First by looking at how a single MCS tends towards its result depending on the number of iterations. Second by looking at how the number of iterations affects the convergence of multiple MCSs towards a similar result.

5.2.1 Internal convergence

We call *internal convergence* the evolution of a given MCS's results as iterations are added to its execution. To observe internal convergence we performed an MCS of OMSSA with 5000 iterations and collected the results of every individual deterministic simulation, called output sample, within the MCS. Using these output sample we are capable of computing the resulting output distribution for any number of iteration below 5000. This effectively simulates what would the result have been if we had stopped the MCS at any arbitrary number of iterations before 5000.

Figure 5.3 shows the evolution of the makespan distribution resulting from an MCS as the number of iterations (N) increases. We can see that in the early stages of the MCS the resulting distribution evolves rapidly:

- At $N = 10$ the distributions appear as high peaks. Values are highly concentrated, but already somewhat asymmetrical with both heuristics displaying a bulge toward higher makespan values.

- At $N = 100$, the distributions do not appear completely normal. The ASAP distribution, in yellow, has a second peak at 13500s. The AFAP distribution, in orange, is made asymmetrical by a plateau around 13850s.

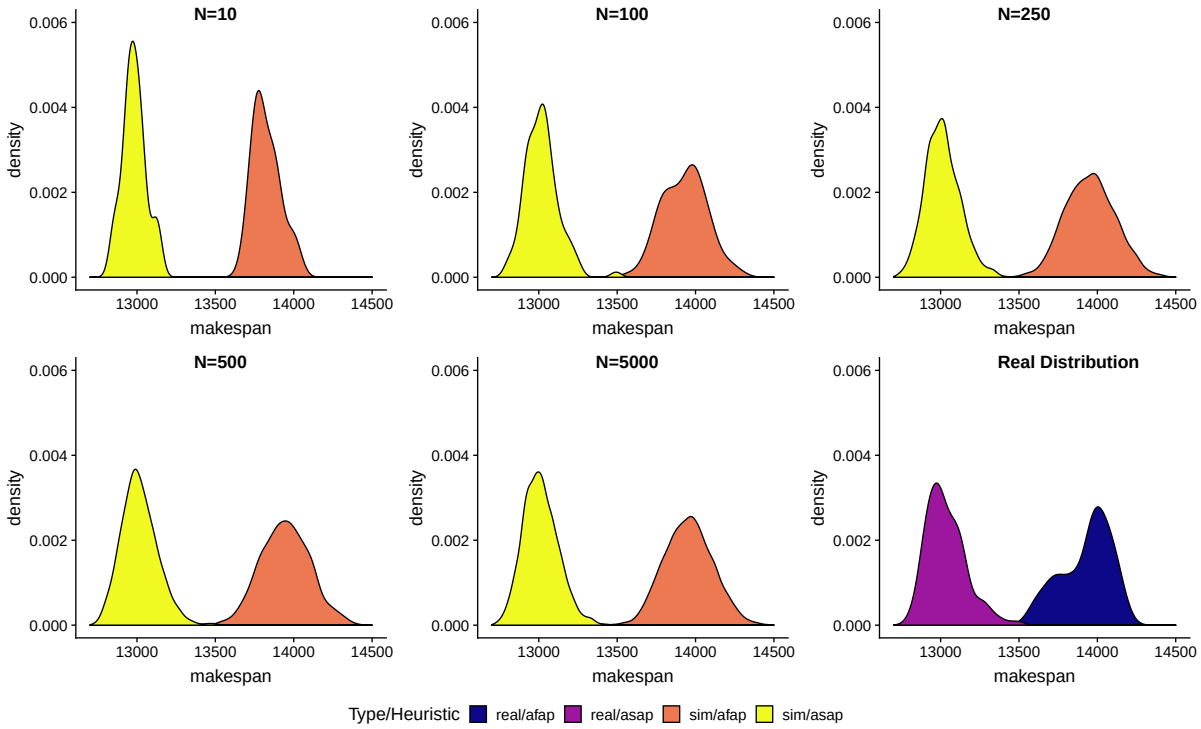

Figure 5.3: Output distributions of an MCS with $P = 10\%$ as the number of iterations N varies.

- At $N = 250$, the plateau observed in AFAP is mostly gone, leaving only a slight asymmetry in the distribution. In ASAP the second peak has started to fade as the main peak gets wider. The samples responsible for the apparition of the second peak at $N = 100$ are still part of the distribution, but of the 150 samples added between $N = 100$ and $N = 250$ had too few makespans around 13500s. Therefore the relative weight of these observations has steadily decreased, as shown by the peak becoming smaller in the density graph.

- At $N = 500$, both distributions now look fairly normal.

The distributions are far less variable past this point. We can still see the shape of the distribution evolve slightly between $N = 500$ and $N = 5000$ but the position of the peaks, which represent μ, and the overall width of the distributions, linked to σ, does not appear to change significantly. Therefore the MCS's CIs and capture rates are not expected to evolve much.

Figure 5.4 shows the evolution of the value of μ found by the MCS as the number of iterations gets larger. The plots on left-hand side present the evolution of μ, the average makespan obtained by the MCS's fitting process, whereas the ones on the right-hand side show the evolution of σ, the standard deviation obtained by the fitting process. All values are provided as percentages of the values obtained when fitting is performed with all 5000 iterations. A horizontal guideline is placed at 100%. We can see the evolution profile varies depending on the strategy used. As in our previous observations, we see results vary rapidly at the start of the simulation when only a small number of iterations have

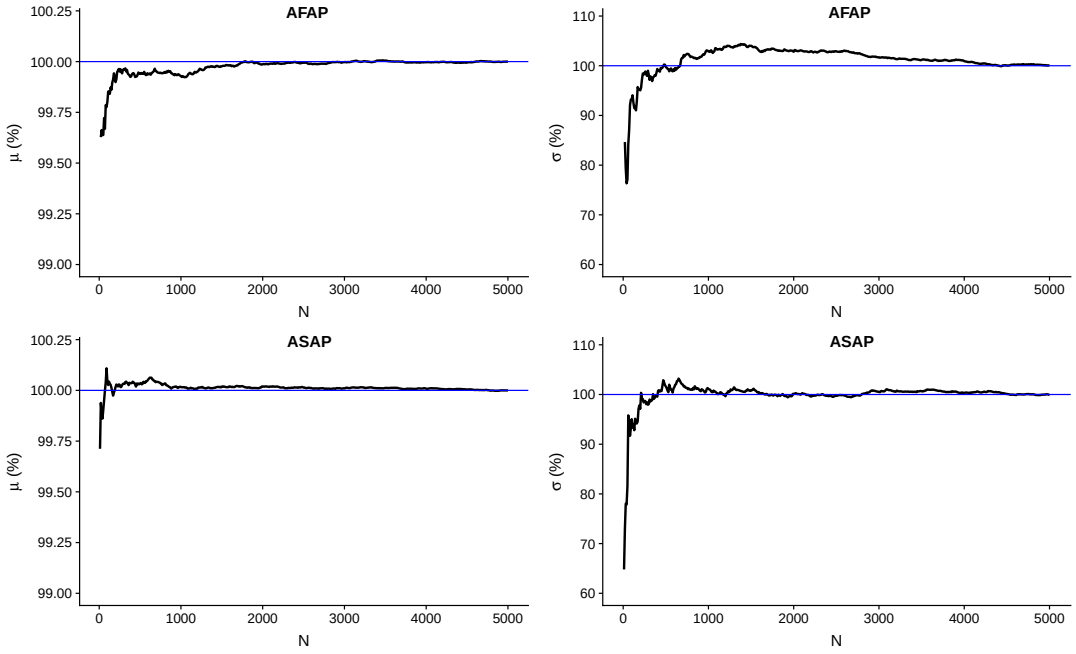

Figure 5.4: Evolution of the estimated values of μ and σ as iterations are performed. Values are given as a percentage of the final value reached when $N = 5000$
Reading example: In the AFAP MCS (first row), once we reach 100 iterations the value of μ generated will always fall within 0.25% of the value found at 5000 iterations.

been performed. The variability slows down significantly as more iterations are added, and the value tends towards the MCS final result. Comparing the results of fitting performed at $N = 500$ and $N = 5000$ we found:

- For AFAP, the value of μ at 500 iterations is within 0.001% ($\approx$ 8s) of the value found using 5000 iterations. The value of σ at 500 iterations is within 0.25% ($\approx$ 0.5s) of the value obtained at 5000 iterations.

- For ASAP, the value of μ at 500 iterations is within 0.03% ($\approx$ 5s) of the value found using 5000 iterations. The value of σ at 500 iterations is within 1.6% ($\approx$ 2s) of the value obtained at 5000 iterations.

Even small numbers of iterations provide values of μ obtained within 1% of the one obtained at 5000 iterations. Values of σ however are more volatile, falling as far as 70% of the final value in the early iterations. Although this volatility diminishes past 1000 iterations, the obtained values of σ can be quite distant from the value found with 5000 iterations. This is notably the case with our MCS in AFAP. In this case the MCS only achieved a value of σ consistently within 1% of the final value once 4000 iterations had been passed.

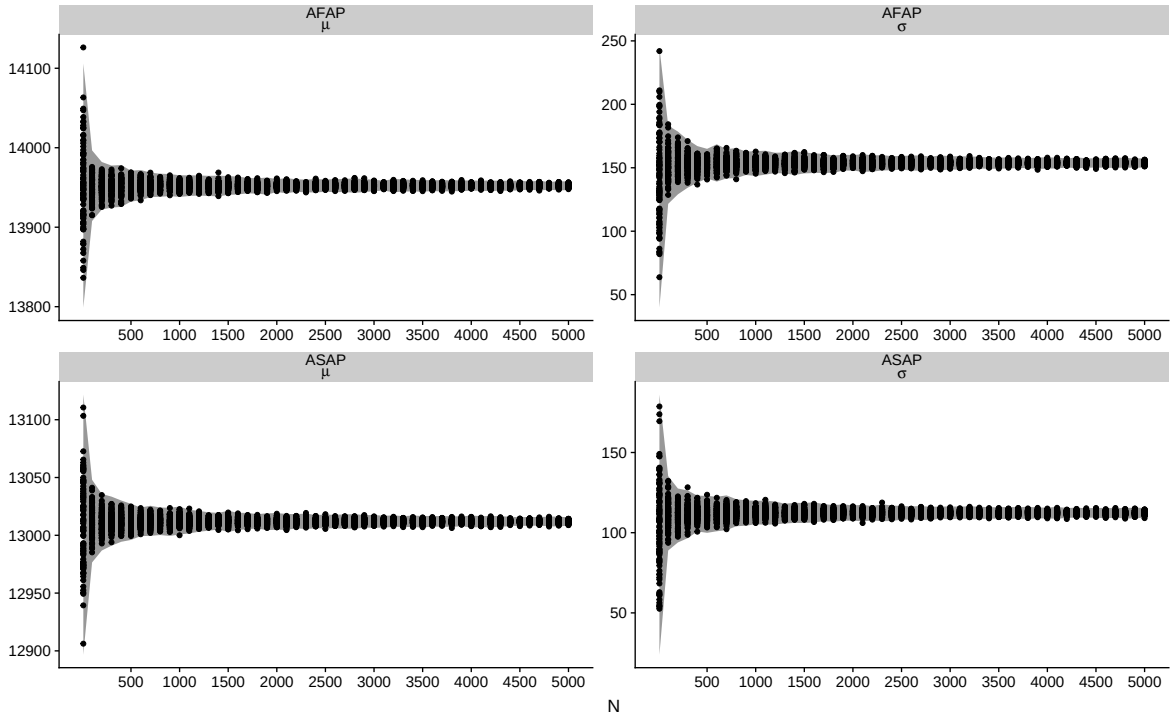

Figure 5.5: Values of μ and σ obtained by different MCS of OMSSA depending on the number of iterations. The provided envelope is the empirical 3 standard deviations range obtained from the plotted data points. All MCSs are of OMSSA with $P = 10\%$
Reading example: Looking at the top left panel, when performing MCSs of 1400 iterations using AFAP, μ ranges from 13939.06s to 13968.81s. This last value is an outlier that falls outside the empirical envelope.

5.2.2 Inter-simulation convergence

Since MCSs are a random process, the results of an MCS can vary from one execution to another. We call *inter-simulation convergence* the evolution of multiple MCS results as iterations are added to each of them. To observe inter-simulation convergence we performed an MCS of OMSSA with 15000 iterations. By sampling the large pool of deterministic simulations by this MCS we are able to generate multiple different *simulated MCSs* with any number of iterations bellow 15000. The number of drawn samples is the number of iterations in our simulated MCS and re-sampling will generate a different simulated MCS.

Using this method we generated 100 simulated MCSs of 10 iterations, 100 simulated MCSs of 100 iterations, and 100 simulated MCSs for every 100 iterations until we reached 5000 iterations. For each of these simulated MCSs we fitted the resulting makespans on a normal distribution and tracked the obtained values for μ and σ.

Figure 5.5 presents the obtained values depending on the number of iterations in a given MCS. For each line of MCSs possessing the same number of iterations we measured the mean resulting value and the standard deviation between MCSs. We used those results to draw an envelope ranging 3 standard deviations around the mean. If the MCSs

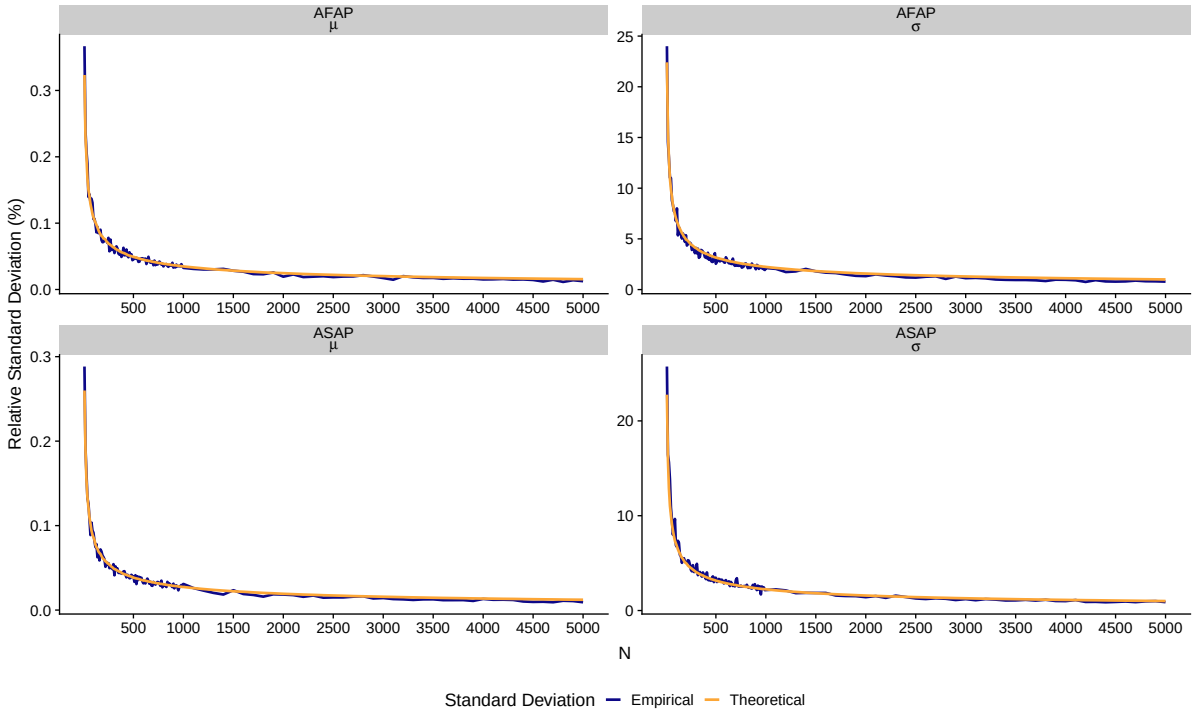

Figure 5.6: Evolution relative to the number of iterations of the empirical standard deviation, measured between MCSs, and the median theoretical standard deviation, obtained by the fitting process. All MCSs are of OMSSA with $P = 10\%$. *Reading example: looking at the relative standard error on σ when using AFAP, top right panel, we can see that the standard deviation reaches 5% of the value for N around 200.*

are normally distributed[1] then 99% of MCSs are expected to fall within the envelope.

The results seen here match what we expect from studying the evolution of the internal convergence of an MCS. The high volatility observed in the early iterations of an MCS results in a high disparity of results between MCSs with low iteration counts. As the number of iterations gets higher every MCS's results stabilise toward a common value. As with internal convergence, this inter-MCS convergence happens very quickly for low iteration counts but becomes asymptotic as the number of iterations augments.

It is worth noting that users interested in the evolution of this inter-MCS convergence do not need to perform thousands of MCSs as we did. The fitting process used at the end of each MCS to estimate the values for μ and σ also produces a margin of error. This margin of error is closely linked to the number of samples provided to the fitting process, i.e. the number of iterations in the MCS. The margin of error is given in the form of a standard deviation on the estimated values of μ and σ.

Figure 5.6 compares the evolution of the theoretical standard deviation on μ and σ, obtained by the fitting process, to the empirical standard deviation measured by comparing multiple MCSs. Although the empirical standard deviation is more noisy than the theoretical one, both match quite closely. This makes standard deviation obtained by

[1]This is different from the result of *an* MCS being a normal distribution.

fitting interesting for building *dynamic* MCSs that add iterations until reaching a given *precision threshold*. It can be computed on the fly by the fitting process of a single MCS and does not require foreknowledge of the final results, like the internal convergence does. The graphs show how these standard deviations on μ and σ evolve similar to the inverse square root of the number of iterations. This evolution of the error profile is common in Monte Carlo processes and matches all our previous observations. These graphs show that by the 500th iteration the MCS is out of its quick evolution phase. Reducing by half the standard deviation obtained at $N = 500$ requires an MCS with around 2000 iterations.

The asymptotic nature of the MCS(s) convergence towards the intractable theoretical result of the stochastic simulation means adding iterations will always result in more precise simulation results. However the increase in precision will become increasingly marginal. The selection of the number of iterations N will likely be dependent on external constraints, such as the context in which the simulation is executed and the capabilities of the system running the MCS.

At $N = 500$ we found our simulations take 15 minutes of CPU time. However deterministic simulations within an MCS are independent and can be executed concurrently. By running 10 simulations in parallel and keeping intermediate files in RAM we managed to bring the total MCS runtime to 2 minutes. This constrains cases in which it is advantageous to use MCSs as prediction tools. Users will need to run one MCS per strategy they want to test. Therefore the workload must be long enough that the time spent running the MCSs remains negligible. Our MCS setup could be further improved by making task file generation parallel to the executions of the first SimSchlouder instances, as well as starting result aggregation in parallel to the simulations of the last realisations.

In the case of offline study, where MCSs are used as a tool to explore IaaS clouds or the behavior of cloud applications, users should set up dynamic MCSs. A dynamic MCS will perform additional iterations until it has reached a target precision level. It does so by regularly fitting the iterations already performed to observe the evolution of the standard deviations on the results. These dynamic MCSs can therefore adapt N to the variability of the workload they are simulating. Figure 5.7 shows how P can affect the convergence of the MCS. It shows that the standard deviation on μ will grow with higher perturbation levels. Although with $P = 50\%$ the MCS still enters the asymptotic regime before $N = 500$, it will do so with a higher standard error. This means that it will take many more iterations for the MCS using $P = 50\%$ to reach the same level of error as an MCS with $P = 10\%$. The standard error on σ however appears to be solely dependent on the number of iterations even at higher perturbation levels.

This effect is not limited to P. Any variable susceptible of changing the size of the realm of realisations is liable to affect the rate of convergence of the MCS. In our case such variables include the task's expected walltimes, the number of tasks in the workloads, and the dependencies between tasks. In physics it is common to find MCSs with thousands of random variables requiring hundreds of thousands of iterations to reach the given precision thresholds. Comparatively our simulations of 233 tasks with a 10% perturbation level are quite simple.

In conclusion, although the number of iterations an MCS should perform is largely dependent on the context of the simulation and the time afforded, we would recommend

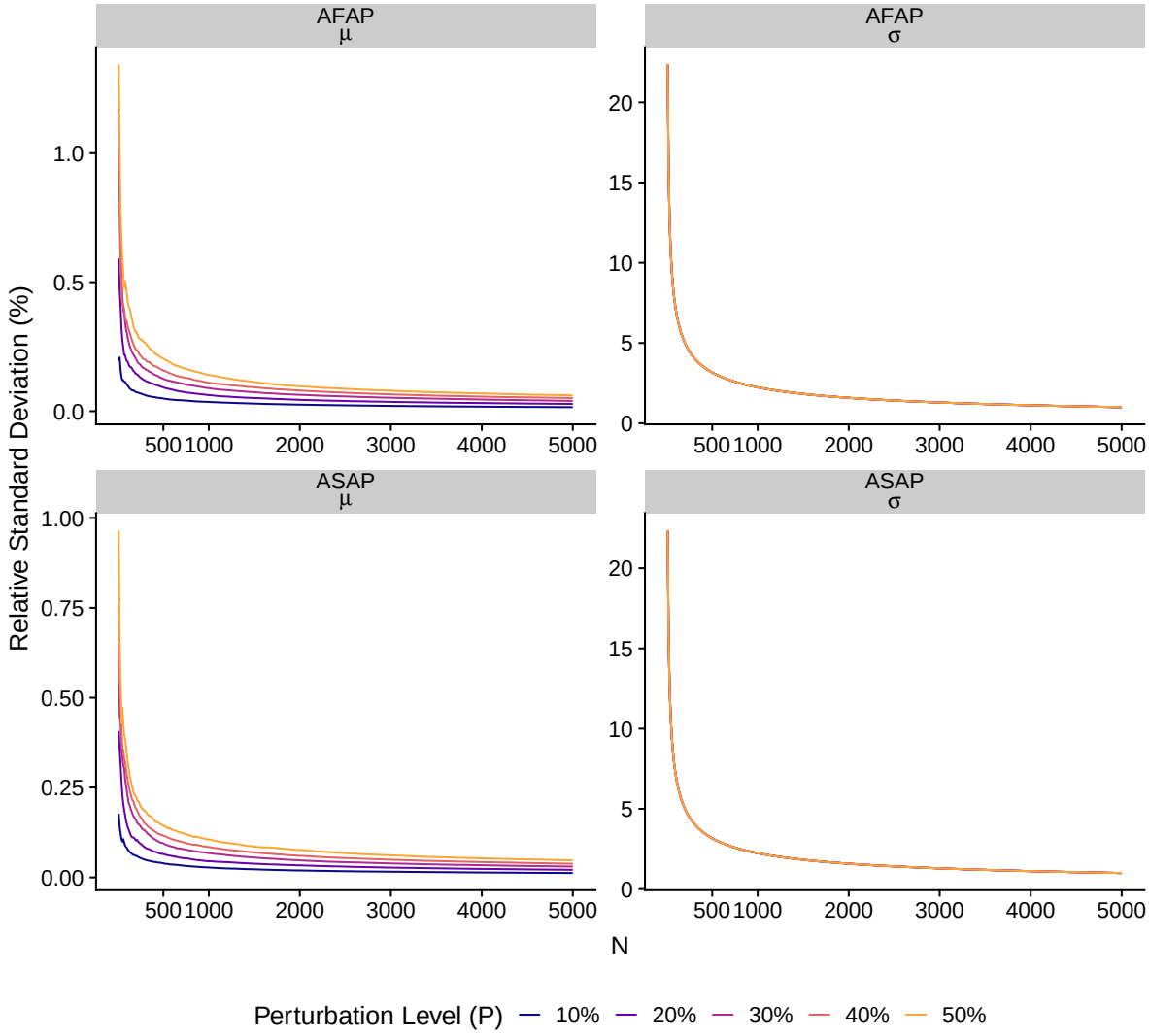

Figure 5.7: Evolution of the margin of error of the fitting process over the course of 5000 iterations at different perturbation levels.

tracking the fitting process's standard deviation as a measure of the level of precision reached.

5.3 Input Distribution Choice

We built our input model around uniform distributions ($\mathcal{U}$). This choice was motivated by the simplicity of usage in contexts in which little information would be available, such as cases where the MCS would be used as a prediction tool.

However doing so impedes the MCS's ability to produce outliers. With a uniform distribution every generated task's walltime is strictly contained within the given interval. Switching to an infinite continuous distribution would allow the MCS to generate on occasion task walltimes arbitrarily far from the expect runtimes providing outliers. The most common such distribution is the normal distribution ($\mathcal{N}$). Because of the central limit theorem, phenomena that are the result of the sum of independent processes often display normal distributions.

We know from our work in Section 4.3.4 that the task walltimes observed in our real executions did not appear to conform to a normal distribution. However these observed walltimes did not conform to a uniform distribution either and our MCS still managed a 90% capture rate. We view a normal distribution as a natural upgrade from our model. Both distributions are symmetrical and parametrized by two variables. The first variable represents the distribution expectancy and the second defines the width of the interval.

The only choice necessary when replacing a previous model with a uniform distribution is fixing the relation between the perturbation level P and the standard deviation necessary for the normal distribution. In our established input model every generated task walltime falls within $P\%$ of the expected value. If we want to produce an input model in which only a few walltimes fall outside of this interval we could choose:

1. $\sigma = P/3$: in this configuration 99% of drawn walltimes would still fall within the previously established interval. This model would only produce 1% of outliers compared to the previously established model.

2. $\sigma = P/2$: in this configuration only 95% of drawn walltimes would fall within the previously established range.

Figure 5.8 compares the makespan distribution of real executions with makespan distributions obtained by MCSs using three different input models. The first uses the uniform distribution, it is the input model first presented in Section 4.3.4. The last two use normal distributions with $\sigma = P/2$ and $\sigma = P/3$ respectively. All simulations were performed at $P = 10\%$.

Although switching to normal distributions did allow for the existence of walltimes beyond the limits imposed with the uniform distribution, the makespan distributions obtained do not reflect that fact. This is the consequence of distribution walltimes within the space that were covered in the uniform input model.

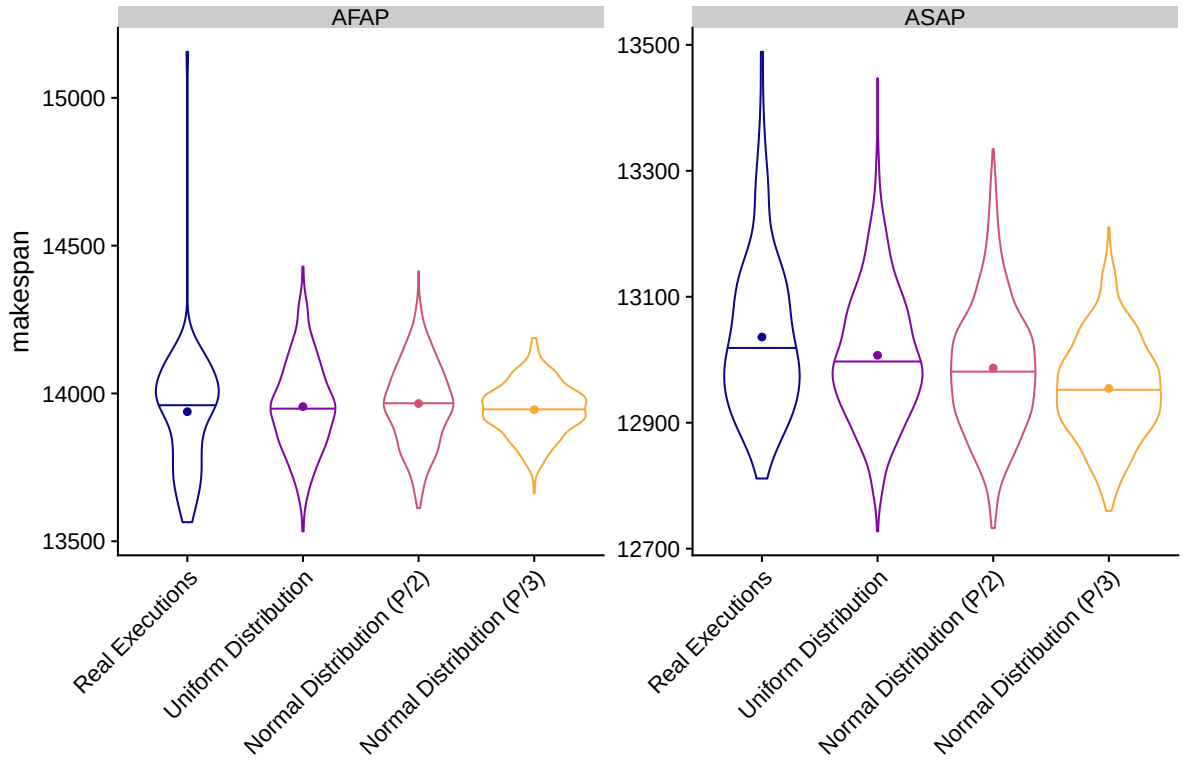

Figure 5.8: Makespan distributions of OMSSA for real executions and MCSs using the uniform input model, normal input model with $\sigma = P/2$ and $\sigma = P/3$. All MCSs are performed with $P = 10\%$. The width of each shape is the probability density. The point indicates the average value. The bar represents the median observation.

		$\mathcal{U}(\mu \pm P\%)$	$\mathcal{N}(\mu, P/2)$	$\mathcal{N}(\mu, P/3)$
CR (%)	AFAP	92.9	91.8	70
	ASAP	90.6	90.6	76.4

Table 5.1: capture rates obtained by the different input models experimented.

In the unifrom input model every possible walltime had an equal opportunity of being drawn. With the normal input model walltimes closer to the expected walltime (μ) are more likely to be drawn than others. This means that even though we can now draw walltimes that are further from the expected walltimes, most of our drawn walltimes will in fact be closer to the expected walltimes. This concentration of values around the expected walltimes is responsible for the contraction for the makespan distributions obtained by the MCS.

Table 5.1 presents the CRs obtained by the different input models. When using $\sigma = P/2$ the normal distribution model almost matches the CR obtained by our uniform distribution model, only capturing one less real execution in AFAP. As expected from Figure 5.8 the normal distribution model using $\sigma = P/3$ produces too short of a confidence interval leading low capture rates.

In this work we placed ourselves in the context of building prediction tools. Because of this we chose our input model to be as generic as possible and to require as little information as possible. There certainly exist values of σ for which a normal distribution input model performs better than our input model. But finding it would require precise understanding of task behavior and amounts to specializing the input model to a given workload. Since in our experience most tasks have a hard limit as to how fast they can execute but no limit on how slowly they can execute users wanting to build highly precise input models should look towards continuous distributions on semi-infinite intervals. However these distributions will often be harder to parametrise, requiring from users more advanced understanding of the distribution and more information about the workload. This is hardly feasible when the MCS is used as a prediction tool, but a real possibility for a user using the MCS in an offline study.

5.4 Take-away

In this chapter we experimented with the variables of our MCSs to see how they affect the simulations' results. In doing so we learned:

- The **perturbation level (P)**. P should be set as close as possible to the simulated environment's variability. Higher CRs can not be achieved by indefinitely increasing the value of P.
- The **number of iterations (N)**. Our MCSs' results displayed low variability past 500 iterations. Although by nature MCSs converge asymptotically, cases like OMSSA and Montage will stabilise within one thousand iterations. Users wanting

to estimate an MCS's convergence should rely on the standard deviation provided by the final fitting process.

- The **input model distribution**. Trying to switch to a normal distribution did not yield any benefits. When using MCSs as a prediction tool the uniform distribution is easier to understand and requires less information.

Chapter 6

The case of MapReduce

After successfully building our MCS for Schlouder, we wanted to take advantage of the MCS's natural extensibility and experiment simulations with different setups. This chapter details our experiment with MCS using MapReduce (MR) workloads.

6.1 MapReduce and Hadoop

MapReduce. MapReduce [16] is a programming model, proposed by Google in 2004, designed for the processing of big data sets in a parallel fashion. The general idea is to apply a same user-provided *map* function to different parts of the data. All execution instances of the map function being independent, they can be run in parallel. The map functions each produce key-value pairs that are collected by the MapReduce framework. The framework then unifies values that share a same key, and presents to a user-provided *reduce* function all unified unique keys with the list of values associated to each of them. The reduce function then processes the list of values to output a result associated to the given key.

Hadoop. Hadoop [54, 1] is an open-source implementation written in Java of the MapReduce programming model. It is composed of common libraries, a distributed file system HDFS, and a MapReduce engine in charge of managing resources and scheduling workloads.

The Hadoop Distributed File System (HDFS) is a distributed data store optimized for MapReduce execution. HDFS is charged with splitting and distributing data across the nodes of the Hadoop cluster. When inserted into HDFS, data is divided into multiple blocks called *splits*, and these splits are replicated across disks in different racks of the cluster for sake of reliability. Over the course of operations, an HDFS service continuously checks that the filesystem has a given number of replicas for each block (the default is 3 copies). Administrators of the Hadoop cluster can customize this replication factor and location information, such as the racks in which the nodes are located, to allow for more resilient data distributions.

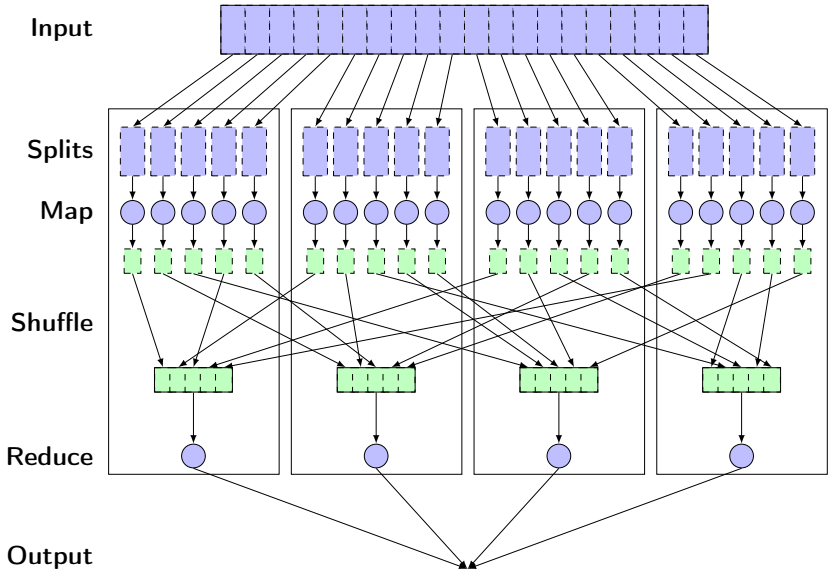

Figure 6.1: Simplified representation of a MapReduce workflow.

Data placement is central to parallel processing in Hadoop. The job manager (YARN) is in charge of the proper operation of the whole workload and also serves as the scheduler. It schedules map and reduce tasks across the infrastructure by starting parallel executions of map function instances for every split of data. For that purpose, the job manager communicates with HDFS to place map tasks on nodes where their specific splits have been distributed. Hence each computation works on local data as the splits are scattered over the nodes' local disks. In cases where this is not possible, the map task will be placed on a node in the same rack as a node containing the split.

The job manager tracks the advancement of each task through a heartbeat mechanism. The workload's final results as well as logs from every task, including the job manager, are usually committed to HDFS at the end of execution.

A typical MapReduce workflow, as shown in Figure 6.1, executes in the following way:

- The workflow's input data is divided into multiple *splits* which are distributed between the different nodes of the MapReduce infrastructure, in a redundant fashion.
- Nodes apply a user-provided *map* function on each input split individually. Results are associated to output keys and written to a temporary storage. A scheduler insures that each node processes local splits in priority and that only one copy of each redundant split is processed.
- During the *shuffle*, map results are redistributed and regrouped based on the output keys produced by the map functions. For specific applications the shuffle can be user-provided.
- Nodes apply the user-provided *reduce* function on each output key it is in charge of.
- Outputs from the different reduce operations are regrouped in the final output(s).

A Hadoop application might be composed of several iterations of this workflow, the outputs of reduce becoming the inputs for the subsequent map functions. For example, iterative graph algorithms, such as the Google's PageRank algorithm [8], adopt this structure to iterate map and reduce phases until a convergence threshold has been reached.

6.2 MapReduce simulator MRSG

To build an MCS we need a deterministic simulator for MapReduce workloads. Building a new simulator from the ground up was not an option considering the time commitment required and we decided to use a pre-existing simulator.

We settled on MRSG (MapReduce over SimGrid) [25] a toolkit for building MapReduce simulations with SimGrid. MRSG is designed to simulate Hadoop version 1. Notice that starting from Hadoop 2, the functionalities of resource management and job scheduling and monitoring have been isolated into a separate service called YARN. Such a change in the Hadoop implementation might modify the experimental results observed with Hadoop 1.

We chose the MRSG simulator because working with a SimGrid-based simulation is a known quantity and the validation of MRSG performed in [25] matches the one we did for SimSchlouder. The MRSG authors report evaluation results carried out on different cluster set ups from the Grid5000 testbed (described hereafter). The cluster set ups used are typically 32 and 64 nodes with 2 cores per node. The evaluation involves three applications shipped with the Hadoop distribution: Log Filter, Tera Sort, and Wordcount. These benchmark applications were run 30 times on the real platform and compared to the simulation. The evaluation of the simulator accuracy is a qualitative one, presented as comparative graphs showing the number of active map and reduce tasks at each second of the execution (presumably the number averaged over the 30 runs) on the one hand and in the simulation on the other. The evaluation does not include comparisons regarding the accuracy of execution time predictions.

6.2.1 Technical specificities

MRSG was designed to help users create simulations of MapReduce workflows. Using SimGrid as the simulation core allows MRSG to efficiently execute large scale simulations. Simulations produced by MRSG reproduce the behavior of the Hadoop implementation of MapReduce. Like SchIaaS, MRSG makes use of the MSG interface of SimGrid to instrument the simulation, but relies on a C API instead of the Java one.

MRSG simulations reproduce the heartbeat mechanism present in Hadoop, it is however unable to fully reproduce the fault recovery behaviors implemented in Hadoop. The data distribution is abstracted through a matrix mapping splits to their respective nodes. This allows MRSG to fully reproduce the data-locality based scheduling observed in Hadoop, while keeping the data splitting out of the scope of the simulation. This matches real world behavior where data is split when submitted to HDFS and not at the execution

of the workload. Although MRSG provides the Hadoop data distribution algorithm, users can provide an alternative algorithm for their simulations. Likewise MRSG implements by default the Hadoop task scheduling but provides the option of using a user-defined scheduler.

In lieu of the actual *map* and *reduce* functions users are asked to provide so called cost functions. During the simulation these functions are called by the simulator to determine the task size in MFlops. An additional function must be provided to indicate the amount of intermediary data generated by each *map* task.

6.2.2 Usage

To build a simulator using MRSG users must provide:

- A task cost function. Taking into account the simulation phase (Map or Reduce), the task id and the node id, this function provides the computational size (in MFlops) of a given task.
- A map output function. Taking into account the originating map task and the receiving reduce task, this function provides the amount of intermediary data emitted by the map tasks.
- (Optionally)A data distribution function. Fills the data distribution matrix based on the number of splits, the number of nodes, and the number of redundant replicas required.
- (Optionally)A scheduler function. Taking into account the simulation phase and the node on which the task will be committed, this function selects the next task to be executed.
- A SimGrid platform file. Used to instantiate the simulated environment.
- A SimGrid deployment file. Used to select the nodes that are part of the MapReduce cluster.
- A MRSG configuration file. Contains all additional information necessary to conduct the simulation. This includes the number and size of input splits, the number of replicas required for data redundancy, the maximum number of simultaneous map tasks, and the number of reduce tasks.

To properly execute a simulation the simulator must first initialise the MRSG library, then set the user-defined functions for task costs, intermediary data size, and if required user-defined function for data distribution and task scheduling. Finally the simulator calls the `MRSG_main` and passes the platform, deployment, and configuration files. The `MRSG_main` function will instantiate the simulated environment and run the simulation.

One important restriction of MRSG is its inability to simulate MR applications composed of several map-reduce phases. We discovered this restriction during the course of

our expriments and the MRSG authors confirmed that extending it to multiple MR phases was not a trivial workaround. This limits the range of examples we can use, although the experiment with a single-phase MR program presented in next section brings useful insights to extrapolate the behavior of multiple-phase MR applications.

6.3 Experiment

Using Hadoop MapReduce and MRSG we intend to perform the same experiment as in Section 4.3. To do so we must generate a body of real executions of a given workload, then execute an MCS reproducing the same workload, and finally compare the distribution obtained by the MCS to the one obtained by the real executions. In this section we will first present the real executions and then the corresponding MCS. Finally, we discuss the results of the comparison.

6.3.1 Real executions

Setup

The workload we choose to execute is TeraSort. TeraSort is a workload provided within the Hadoop distribution, wildly used to benchmark MapReduce platforms.

TeraSort is a MapReduce workload designed to sort an array of 100-byte long rows. Although TeraSort can be executed on any properly formatted input, Hadoop provides TeraGen a MapReduce application designed to generate such an array directly in HDFS. A third workload TeraValidate processes the TeraSort results to validate whether the results are properly sorted.

To execute these workloads we established MapReduce clusters on Grid'5000[1] [6]. Grid'5000 is a large scale testbed for experimental research in computer science developed under the INRIA ALADDIN development action with support from the CNRS, RENATER, and several Universities as well as other funding bodies. Grid'5000 offers the possibility to users to provision nodes from different clusters made available by the different locations participating in the Grid. The executions used in this experiment were performed on the *graphene* cluster found in the Grid'5000 Nancy location. The *graphene* cluster was chosen because its technical description as a `platform.xml` has been used and validated in other experimental work with SimGrid.

The *graphene* cluster is composed of 131 nodes. Each node contains a single Intel Xeon X3440 2.53GHz quadri-core CPU. Nodes are connected through a 1Gb/s Ethernet network and 20 Gb/s InfiniBand network. `hadoop_g5k.py`[2], a script written by Miguel Liroz, allowed us to easily deploy a Hadoop cluster on the provisioned nodes, automatically configuring our HDFS with the topology of the cluster.

[1] https://www.grid5000.fr
[2] https://github.com/mliroz/hadoop_g5k

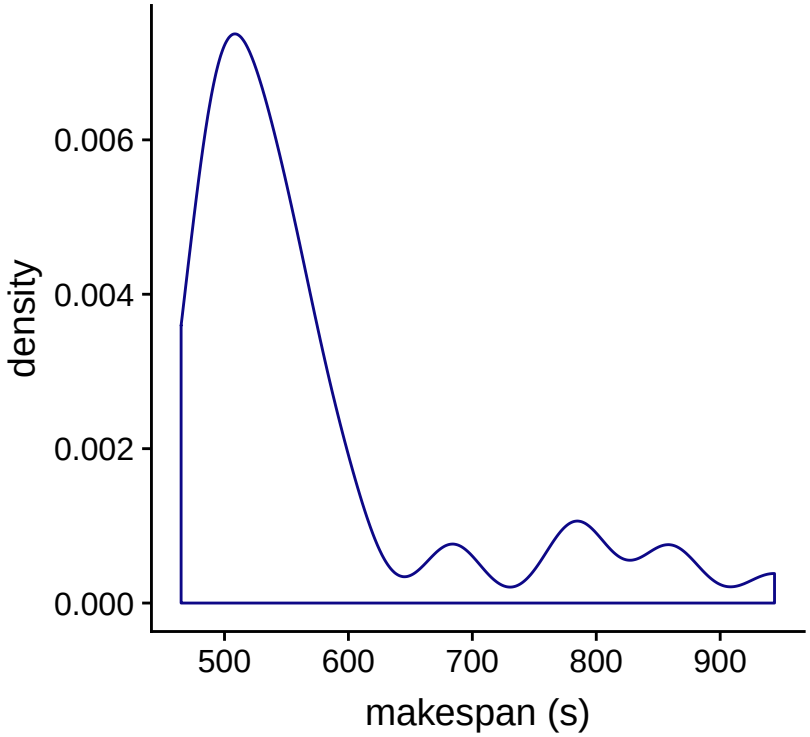

Figure 6.2: Makespan distribution obtained for our executions of TeraSort on the graphene cluster.

Real observations

We executed 45 runs of TeraSort on 16 nodes provisioned from the *graphene* cluster. The executions were performed in two sets, one of 15 executions and the other of 30 executions. Each set was executed on different days (respectively May 25, 2018 and June 13, 2018) on different nodes of the cluster. At the time of the experiment the cluster was almost fully occupied. Within a set all executions of TeraSort are executed on the same TeraGen generated array. The array produced contains one billion lines for a total size of 100Gbytes.

Figure 6.2 shows the distribution of makespans obtained during our execution of TeraSort. We can see that most of our executions (82.2%) last less than 11 minutes ($660s$). These makespans taken separately show a fairly normal distribution. The remaining 8 executions (17.8%) all present higher makespan values. This asymmetry is typical of the variability of resource contention in distributed systems, there is a hard lower limit as to how fast a workload can be executed, but when the platform is overloaded there is no limit as to how long the workload execution can take. After having investigated the tasks that have greater deviations from the mean, it appears that they are not correlated to a specific split indicating this is not a bias of the Hadoop implementation. Further, these outliers are not found in a particular run of the batch but appear in different runs. Hence, this makespan distribution indicates that some executions were subjected to higher cross-traffic than the others.

6.3.2 Simulation

Setup

Since MRSG allows us to control the computational size of a task through a function executed at runtime our setup does not require generating task size through a script beforehand. Instead the random number generation is handled inside the task cost function at runtime. Doing so required us to touch the code of MRSG to prevent it from seeding the random number generator with a constant. To guarantee MRSG simulations would assign splits to nodes in a consistent fashion the developers had fixed the random number generator seed. In our MCS consistent split distribution is neither necessary, nor wanted.

The simulator thus built, called `rterasort`, is the core of our new MCS. Since `rterasort` does not require pre-generating task files, new iterations can simply be obtained by executing `rterasort`. Because none of the additional features `lab.py` provides are necessary for the simulation, it is instrumented by an ad-hoc shell script.

Input model

We build our input model using the same method as the one presented in Section 4.3.4. We set our perturbation level by first comparing tasks working on the same input to find the worst relative deviation from the mean and then averaging the worst deviations across all tasks (Equation 4.3 page 54). On our first set of executions, the one with of 15 executions, this method yields a perturbation level of 19.5%. On our second set, containing 30 executions, this method yields a perturbation level of 18.1%. Our MCS will use $P = 20\%$.

To check our hypothesis that our execution set's variability can be partially attributed to platform charge, we ran a third set of executions in lighter conditions. Since graphene has been taken out of production this new set has been executed on the *chetemi* cluster in the Grid'5000 Lille location on May 7, 2019. From the monitoring tools of Grid5000 we established that the external charge, defined as the sum of CPU times used by other users (using 4 nodes out of the 15) during our two-hour experiment, was 16% of the total CPU time used. This set of executions resulted in a perturbation level of 10.1%.

For the expected time values the nature of using a cost function makes it complicated to use a different value for every single map task. However looking at the execution traces we can see that the map tasks can be separated in 3 groups depending on input size. Therefore our cost function will use three different expected walltimes obtained by averaging the observed walltimes in each of the three groups. For the purpose of `rterasort` the cost function will convert the generated walltime into a computational size by using the nodes' simulated speeds.

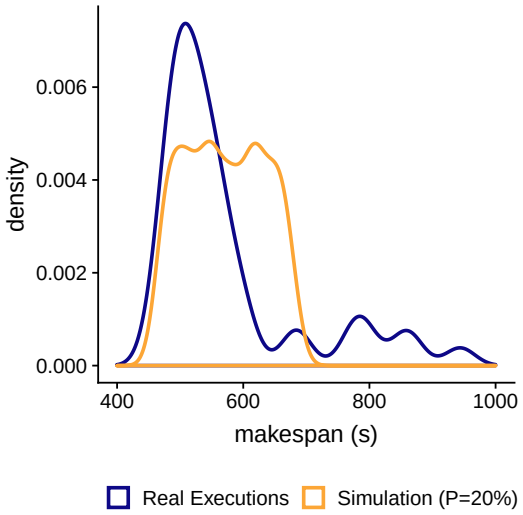

Figure 6.3: Makespan distirubtions obtained by our MCS and by our real executions. *Reading Example : The simulation fails to capture the two fastest real executions under* 467*s and the slowest executions over* 670*s*.

Simulation results

Figure 6.3 shows the makespan distributions obtained by our MCS and by our real executions. The results are quite different from what we observed with our MCSs of Schlouder in Section 4.3.5. The MCS results do not appear to be normal like the results of our previous MCS.

For the capture rate, using the same normal fitting as in Section 4.3.5, the 95% CI gives a capture rate of 86.7%. However the simulated distribution appears to be a uniform distribution, instead of a normal one, and the capture rate between the minimum simulated walltime and the maximum simulated walltime is of 77.8%.

6.3.3 Discussion

The results obtained in this experiment highlight some of the limits of our MCS approach and our input model.

First, our input model is built using uniform distributions centered around expected runtimes. This assumes a symmetrical distribution, this worked well in cases where the overall platform contention is stable. However in this case we observe intermittent additional variability that our model does not capture correctly. Capturing this kind of variability requires an input model based on asymmetrical distributions.

Second, we used in this experiment the exact same method as in Section 4.3.4 to set the perturbation level. This method averages the normalized most extreme outliers of each task group. Like with OMSSA we grouped tasks by inputs, instead of simply grouping them based on the Map/Reduce dichotomy. Using the latter grouping method produces a perturbation level of 115%. We saw in Section 5.1 that our method for setting

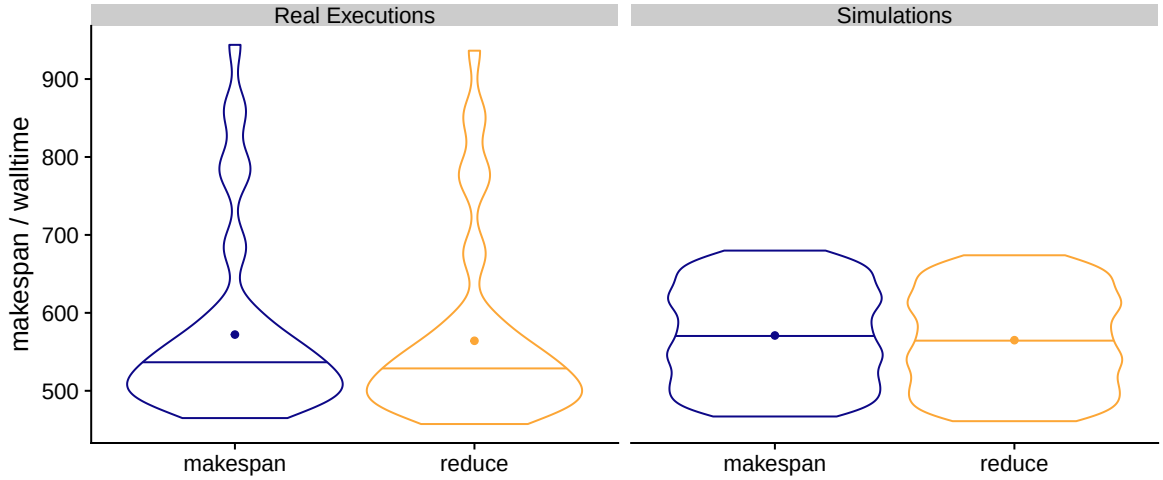

Figure 6.4: Comparison of the makespans and *reduce* task walltime distributions observed in real executions and in our MCS.

the perturbation level does not produce the best possible value of P, another limit seen here is that our approach only works if the tasks are logically split into a high enough number of groups. As the number of groups diminishes only the most extreme outliers get averaged resulting in higher calculated perturbation level. In future work we will look into better methods to estimate P. One such method being considered relies on random sampling of observed walltimes.

Third, the results of the simulations obtained here do not appear to conform to a normal distribution. In our previous experiment we relied on the normal fit of the simulation results to build the confidence intervals from which the capture rate was calculated. The rational presented to justify this choice, beyond the appearance of the simulated distribution, was based on the application of the central limit theorem which states that the distribution of a random variable which is the result of the sum of multiple independent random variables of finite variance will tend towards a normal distribution. In our case we consider the overall simulated makespan as a random variable made from sum of the random variables representing the task walltimes in the critical path. In OMSSA this critical path is at least 55 tasks long. In MapReduce we experience two different effects:

1. **Shallow critical path.** MapReduce assigns tasks on cores rather than nodes or VMs. This makes reaching high levels of parallelism easier. Moreover our 100Gbytes array required only 150 map tasks split between 64 cores. This means our critical path probably contained only 4 tasks. This limits the applicability of the central limit theorem and risks resulting in non-normal distributions.

2. **Dominant reduce task.** In the Hadoop implementation of the MapReduce model, the reduce task is started after 20% of map tasks have finished. Although the reduce function can not be executed until all maps have finished, the intermediary results from the finished maps can be downloaded while the remaining map tasks finish. This shortens significantly the transfer window between the end of last map task

and the start of the reduce task. Because of this the reduce task has a much higher walltime than map tasks. In our executions we observed map walltimes ranging from $2s$ to $12s$ whereas reduce walltimes ranged from $457s$ to $936s$. The difference in order of magnitude makes the variation produced by our MCS on map tasks meaningless compared to the variation produced on the reduce task. In the case of Terasort, we see on Figure 6.4 that, in the real execution and in simulations, the makespan distributions are similar to the reduce distributions.

This is partly caused by our use of TeraSort as a workload. TeraSort uses only one map-reduce phase, with an extremely short map task and a single reduce task. For many other MR applications that involve multiple MR phases, the critical path contains the succession of the reduce tasks, which are of similar size due to the iterative pattern. If the number of iterations is sufficient, the application's makespan is the sum of walltimes considered random variables, and hence makespan values should follow a normal distribution.

This hyptothesis could not be tested in this work because MRSG cannot simulate iterative workflows as noticed earlier (see page 78). Validating this idea in future work will require finding or creating another MapReduce simulator. Our prospective new simulator should also allow us to separate the reduce walltime between the wait time entirely dependent on map tasks' walltimes and the active reduce computation times.

6.4 Take-away

By attempting to build an MCS of MapReduce using the same methods as the one used for Schlouder, we highlighted some of our approach's limits to MCSs.

- There is a limit to the variability our single parameter approach can model, especially concerning very unstable environments. Those require dedicated distributions specifically designed to match the unstable and asymmetric nature of the environment.
- The characterization of a perturbation level to be used with a simple symetric distribution is difficult since its computation is sensitive to the grouping of tasks. Building very large groups results in the overweighing of observed extreme values. In this MR test-case, considering all map tasks to belong to two groups only depending on the split size resulted in a high perturbation level, which in turn yields a makespan prediction interval that is too large to be relevant.
- To take advantage of the stochastic nature of our MCS the simulated workload must be comprised of a sufficiently high number of tasks of the same order of magnitude in size. If one task dominates the workload, we can not take advantage of the compounding nature of the variability introduced by the MCS.

Conclusion

Over the last fifteen years, the increasing ubiquity of internet access and advances in virtualisation technologies have lead to a *Cloud computing* boom. Infrastructure as a Service (IaaS) clouds have profoundly changed the *hosting* landscape. The ability to provision resources on the fly lets users scale their infrastructure precisely to their needs. This in turn makes Platform as a Service (PaaS) and Software as a Service (SaaS) more viable business models. Although Cloud computing has been hugely successful in the web service area, it does not appear to have gained as much traction in the scientific computing area.

There are many reasons not to execute scientific applications in cloud environments. Some applications are designed to be executed on HPC clusters, high computing power and low latency environments, with whom IaaS can not always compete. Others rely on external schedulers for their execution, these are mostly found on institutional grids and clusters. The *pay-as-you-go* model can also lead to budgetary uncertainty, making using the cloud a higher risk.

Schlouder was designed to provide a grid-like scheduling on IaaS clouds. It combines the familliar interface of a batch-job queuing system with the advantages provided by the on-the-fly provisioning of IaaS. However the effectiveness of different scheduling strategies proved to be extremely variable depending on the scientific application being scheduled. This prompted the need for a prediction module capable of providing users with insights about the different scheduling heuristics' behaviors depending on the submitted applications.

Such a prediction tool requires being able to precisely simulate the execution of a workload by Schlouder. For this purpose we developed SimSchlouder based on the SimGrid and SchIaaS toolkits. By comparing over 336 real execution traces to their respective simulations, we were able to isolate the information necessary to establish precise simulations. The variability observed between identical executions makes it impossible to simulate precisely a workload without first executing it to measure it precisely.

Our thesis is that by using stochastic simulations we can address the variability observed in real executions and provide more pertinent simulations that require less inputs. Using a simple input model we can represent the variability of a system and absorb part of the imprecision of our expected walltimes. Monte Carlo simulations allow us to easily build stochastic simulations out of our well-tested deterministic simulator SimSchlouder.

To put our thesis to the test we developed a pipeline to compare our MCSs with a

body of real executions. Concurrently we created our input model that relies on already available expected walltimes and a single *perturbation level* that represents the variability of the whole system. Our evaluation shows that our MCS captures upwards of 90% or real executions.

We studied how different parameters effect the results of our MCS. Contrary to our early hypothesis the perturbation level can not be used to trade higher capture rates against larger confidence intervals. There is an optimal value for which the perturbation level most closely represents the variability of the real system and the MCS obtains the highest capture rate. We observed that the MCS converges in an asymptotic manner toward its final result and that the number of iterations necessary to reach a given precision threshold is dependent on the perturbation level. We experimented with using normal distributions in our input model, highlighting the difficulties posed by using more complex distributions.

Finally we tried to build an MCS of the MapReduce workload TeraSort. The difficulties encountered in this experiment are indicative of areas where more research is needed. Firstly, we observed that our approach can absorb input imprecision only when the simulated workload is composed of a large enough number of balanced tasks. Secondly, this case study exhibits more irregular forms of variability and the next challenge will be to create input models that are better at representing them. These advanced models would require an offline analysis of real execution traces to set the right model parameters. The challenge posed by such analysis is bridging the gap between the modeled distribution and the real distribution as we observe it.

MCSs allow us to obtain more pertinent results with less information. In cases where MCSs are used to inform operational decisions, such as the platform sizing or a scheduling heuristic choice, a coarse bounding of the task performance is enough. However users are still required to know enough about their applications and platform to parametrise the input model.

This study of the MCSs' usage, parameters, and limits lays the groundwork for more widespread usage of stochastic simulation around IaaS clouds. Although we are proud of the results we achieved in a prediction context, more interesting uses of MCSs lie outside this limited scope. MCSs can be used study more precisely the interaction between variability, scheduling, and workflows. This opens the door to the creation of scheduling meta-strategies that can adapt their scheduling decisions to the experience variability and the shape of the workflows. Or by using stochastic provisioning patterns, a cloud operator could use MCSs to develop and validate a VM placement algorithm that maximises energy efficiency without compromising on preexisting contractual obligations of responsiveness. Monte Carlo simulations are a promising lead towards variability-aware systems for cloud platforms.

References

[1] *Apache Hadoop*. URL: https://hadoop.apache.org/.

[2] *AWS Auto Scaling*. URL: https://aws.amazon.com/autoscaling/.

[3] Luke Bertot, Stéphane Genaud, and Julien Gossa. "An Overview of Cloud Simulation Enhancement using the Monte-Carlo Method". In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing CCGRID*. 2018.

[4] Luke Bertot, Stéphane Genaud, and Julien Gossa. "Improving Cloud Simulation Using the Monte-Carlo Method." In: *Euro-Par 2018, LNCS 11014*. 2018.

[5] Luke Bertot, Julien Gossa, and Stéphane Genaud. "Méthode pour l'étude expérimentale par la simulation de clouds avec SCHIaaS." In: *Compas'17*. 2017.

[6] Raphael Bolze et al. "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed". In: *IJHPCA* 20.4 (2006), pp. 481–494. DOI: 10.1177/1094342006070078. URL: https://doi.org/10.1177/1094342006070078.

[7] *BonFire*. URL: http://bonfire-project.eu/home.

[8] Sergey Brin and Lawrence Page. "The Anatomy of a Large-scale Hypertextual Web Search Engine". In: *Comput. Netw. ISDN Syst.* 30.1-7 (Apr. 1998), pp. 107–117. ISSN: 0169-7552. DOI: 10.1016/S0169-7552(98)00110-X. URL: http://dx.doi.org/10.1016/S0169-7552(98)00110-X.

[9] Zhicheng Cai, Qianmu Li, and Xiaoping Li. "ElasticSim: A Toolkit for Simulating Workflows with Cloud Resource Runtime Auto-Scaling and Stochastic Task Execution Times". In: *J. Grid Comput.* 15.2 (2017), pp. 257–272. DOI: 10.1007/s10723-016-9390-y. URL: https://doi.org/10.1007/s10723-016-9390-y.

[10] Rodrigo N Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and experience* 41.1 (2011), pp. 23–50.

[11] Louis-Claude Canon and Emmanuel Jeannot. "Evaluation and Optimization of the Robustness of DAG Schedules in Heterogeneous Environments". In: *IEEE Trans. Parallel Distrib. Syst.* 21.4 (2010), pp. 532–546. DOI: 10.1109/TPDS.2009.84. URL: http://dx.doi.org/10.1109/TPDS.2009.84.

[12] Christine Carapito et al. "MSDA, a proteomics software suite for in-depth M ass S pectrometry D ata A nalysis using grid computing". In: *Proteomics* 14.9 (2014), pp. 1014–1019.

[13] Eddy Caron, Frederic Desprez, and Adrian Muresan. "Forecasting for Grid and Cloud Computing On-Demand Resources Based on Pattern Matching". In: *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*. CLOUDCOM '10. IEEE Computer Society, 2010, pp. 456–463. ISBN: 978-0-7695-4302-4. DOI: 10.1109/CloudCom.2010.65.

[14] Henri Casanova et al. "Versatile, scalable, and accurate simulation of distributed applications and platforms". In: *J. Parallel Distrib. Comput.* 74.10 (2014), pp. 2899–2917. DOI: 10.1016/j.jpdc.2014.06.008. URL: http://dx.doi.org/10.1016/j.jpdc.2014.06.008.

[15] Trieu C Chieu et al. "Dynamic scaling of web applications in a virtualized cloud computing environment". In: *2009 IEEE International Conference on e-Business Engineering*. IEEE. 2009, pp. 281–286.

[16] Jeffrey Dean and Sanjay Ghemawat. "Mapreduce: Simplified data processing on large clusters, 2004". In: *OSDI: Sixty Symposium on Operating System Design and Implementation*. 2004.

[17] Bajis Dodin. "Bounding the project completion time distribution in PERT networks". In: *Operations Research* 33.4 (1985), pp. 862–881.

[18] Brian Dougherty, Jules White, and Douglas C. Schmidt. "Model-driven auto-scaling of green cloud computing infrastructure". In: *Future Generation Computer Systems* 28.2 (2012), pp. 371–378. ISSN: 0167-739X. DOI: 10.1016/j.future.2011.05.009.

[19] Ta Nguyen Binh Duong, Xiaorong Li, and Rick Siow Mong Goh. "A Framework for Dynamic Resource Provisioning and Adaptation in IaaS Clouds". In: *CloudCom'11*. 2011, pp. 312–319.

[20] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*. IEEE Computer Society, 2015, pp. 171–172. ISBN: 978-1-4799-1957-4. DOI: 10.1109/ISPASS.2015.7095802. URL: https://doi.org/10.1109/ISPASS.2015.7095802.

[21] L. Y. Geer et al. "Open mass spectrometry search algorithm". In: *J Proteome Res.* 3.5 (Sept. 2004), pp. 958–964.

[22] Joseph C Jacob et al. "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking". In: *International Journal of Computational Science and Engineering* 4.2 (2009), pp. 73–87.

[23] In Kee Kim, Wei Wang, and Marty Humphrey. "PICS: A Public IaaS Cloud Simulator". In: *8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015*. Ed. by Calton Pu and Ajay Mohindra. IEEE Computer Society, 2015, pp. 211–220. ISBN: 978-1-4673-7287-9. DOI: 10.1109/CLOUD.2015.37. URL: https://doi.org/10.1109/CLOUD.2015.37.

[24] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. "GreenCloud: a packet-level simulator of energy-aware cloud computing data centers". In: *The Journal of Supercomputing* 62.3 (2012), pp. 1263–1283.

[25] Wagner Kolberg et al. "MRSG - A MapReduce Simulator over SimGrid". In: *Parallel Computing* 39.4-5 (Apr. 2013), pp. 233–244. DOI: 10.1016/j.parco.2013.02.001. URL: https://hal.inria.fr/hal-00931855.

[26] *KVM - Kernel Virtual Machine*. URL: https://www.linux-kvm.org/page/Main_Page.

[27] Philipp Leitner and Jürgen Cito. "Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds". In: *ACM Trans. Internet Techn.* 16.3 (2016), 15:1–15:23. DOI: 10.1145/2885497. URL: http://doi.acm.org/10.1145/2885497.

[28] Philipp Leitner et al. "CloudScale: a novel middleware for building transparently scaling cloud applications". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, pp. 434–440.

[29] Yan Alexander Li and John K. Antonio. "Estimating the execution time distribution for a task graph in a heterogeneous computing system". In: *6th Heterogeneous Computing Workshop, HCW 1997, Geneva, Switzerland, April 1, 1997*. IEEE Computer Society, 1997, pp. 172–184. ISBN: 0-8186-7879-8. DOI: 10.1109/HCW.1997.581419. URL: http://dx.doi.org/10.1109/HCW.1997.581419.

[30] Seung-Hwan Lim et al. "MDCSim: A multi-tier data center simulation, platform". In: *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*. IEEE Computer Society, 2009, pp. 1–9. ISBN: 978-1-4244-5012-1. DOI: 10.1109/CLUSTR.2009.5289159. URL: https://doi.org/10.1109/CLUSTR.2009.5289159.

[31] *LINPACK*. URL: https://www.netlib.org/linpack/.

[32] Arfst Ludwig, Rolf H. Möhring, and Frederik Stork. "A Computational Study on Bounding the Makespan Distribution in Stochastic Project Networks". In: *Annals OR* 102.1-4 (2001), pp. 49–64. DOI: 10.1023/A:1010945830113. URL: http://dx.doi.org/10.1023/A:1010945830113.

[33] Paul Marshall, Kate Keahey, and Timothy Freeman. "Elastic Site: Using Clouds to Elastically Extend Site Resources". In: *CCGRID'10*. 2010, pp. 43–52.

[34] David Mendez, Mario Villamiazr, and Harold Castro. "e-Clouds: Scientific Computing as a Service". In: *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on*. IEEE. 2013, pp. 481–486.

[35] Etienne Michon et al. "Schlouder: A broker for IaaS clouds". In: *Future Generation Comp. Syst.* 69 (2017), pp. 11–23. DOI: 10.1016/j.future.2016.09.010. URL: http://dx.doi.org/10.1016/j.future.2016.09.010.

[36] Alberto Nuñez et al. "iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator". In: *J. Grid Comput.* 10.1 (2012), pp. 185–209. DOI: 10.1007/s10723-012-9208-5. URL: https://doi.org/10.1007/s10723-012-9208-5.

[37] Simon Ostermann et al. "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing". In: *Cloud Computing - First International Conference, CloudComp 2009, Munich, Germany, October 19-21, 2009 Revised Selected Papers*. Vol. 34. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2009, pp. 115–131. DOI: 10.1007/978-3-642-12636-9_9.

[38] Julien Perez et al. "Multi-objective reinforcement learning for responsive grids". In: *Journal of Grid Computing* 8.3 (2010), pp. 473–492. DOI: 10.1007/s10723-010-9161-0. URL: http://hal.archives-ouvertes.fr/hal-00491560/en/.

[39] Dana Petcu et al. "Experiences in building a mOSAIC of clouds". In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (2013), p. 12.

[40] Alexander Pucher et al. "Using Trustworthy Simulation to Engineer Cloud Schedulers". In: *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*. 2015, pp. 256–265. DOI: 10.1109/IC2E.2015.14. URL: https://doi.org/10.1109/IC2E.2015.14.

[41] *RightScale*. URL: http://www.rightscale.com.

[42] Iman Sadooghi et al. "Understanding the performance and potential of cloud computing for scientific applications". In: *IEEE Transactions on Cloud Computing* 5.2 (2017), pp. 358–371.

[43] *Scalr - The Hybrid Cloud Management Platform*. URL: https://www.scalr.com.

[44] *SCHIaaS: IaaS simulation upon SimGrid*. URL: http://schiaas.gforge.inria.fr.

[45] *Schlouder: IaaS cloud broker for public or private clouds*. URL: https://schlouder.gforge.inria.fr.

[46] *SimGrid: Versatile simulation of distributed systems*. URL: https://simgrid.org/.

[47] *SimSchlouder: Schlouder simulation upon SCHIaaS*. URL: http://schiaas.gforge.inria.fr/simschlouder.html.

[48] Richard M. van Slyke. "Monte Carlo Methods and the PERT Problem". In: *Operations Research* 11.5 (1963), pp. 839–860. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/167918.

[49] Hu Song, Jing Li, and Xinchun Liu. "IdleCached: An Idle Resource Cached Dynamic Scheduling Algorithm in Cloud Computing". In: *9th UIC and ATC*. Sept. 2012, pp. 912–917.

[50] Hans Gerd Spelde. "Stochastische Netzpläne und ihre Anwendung im Baubetrie". PhD thesis. Rheinisch-Westfälische Technische Hochschule Aachen, 1976.

[51] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience". In: *Concurrency - Practice and Experience* 17.2-4 (2005), pp. 323–356. DOI: 10.1002/cpe.938. URL: https://doi.org/10.1002/cpe.938.

[52] Christian Vecchiola et al. "Deadline-driven provisioning of resources for scientific applications in hybrid clouds with Aneka". In: *Future Gener. Comput. Syst.* 28.1 (Jan. 2012), pp. 58–65. ISSN: 0167-739X. DOI: 10.1016/j.future.2011.05.008.

[53] David Villegas et al. "An Analysis of Provisioning and Allocation Policies for Infrastructure-as-a-Service Clouds". In: *CCGRID'12*. 2012, pp. 612–619.

[54] Tom White. *Hadoop: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2009. ISBN: 0596521979, 9780596521974.

[55] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. "URL: A unified reinforcement learning approach for autonomic cloud management". In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 95–105. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2011.10.003.

Annexe A

Amélioration de simulation de cloud IaaS via l'emploi de méthodes stochastiques.

A.1 Motivations

L'évolution rapide des technologies de virtualisation et des réseaux informatiques a fait émerger ces deux dernières décennies de nouveaux modèles d'exploitation des ressources informatiques. Ces nouveaux modèles qu'on regroupe sous le terme générique *Cloud*, ont conduit à une transformation radicale de l'hébergement et du développement en informatique.

L'un de ces modèles est l'Infrastructure as a Service (IaaS). Il rend possible le dimensionnement dynamique de l'infrastructure, en permettant à l'utilisateur (qui devient un client du fournisseur du service d'infrastructure) d'acquérir ou de relâcher des ressources en fonction de l'évolution de ses besoins en calcul et stockage. L'acquisition des ressources sous forme d'une location est généralement facturée proportionnellement au temps d'utilisation. Ainsi l'utilisateur de ressources IaaS est incité à gérer finement la quantité de ressources louées afin d'optimiser ses coûts de fonctionnement.

Si ce modèle est très populaire dans le contexte de l'hébergement web, son utilisation dans le contexte du calcul scientifique recouvre des cas d'usage différenciés. Même si nous excluons de notre discussion le domaine du calcul haute-performance qui fait l'hypothèse d'exploiter des super-calculateurs, il reste un éventail large d'applications scientifiques conçues pour être exécutées sur des systèmes distribués. Les applications de ce type ont largement bénéficié du développement des grilles institutionnelles de calcul au début des années 2000 et sont aujourd'hui des candidats pertinents pour être adaptés au cloud IaaS.

En passant des grilles de calcul au cloud IaaS, le paradigme économique concernant l'exploitation des ressources change. Les coûts d'investissement et d'exploitation d'une grille sont mutualisés, et ne sont généralement pas imputés directement aux utilisateurs individuels. Le cloud, en revanche, généralise le *payement à l'usage*, facturé à chaque

utilisateur individuel. Ceci constitue une très forte incitation pour que les utilisateurs prédisent précisément leurs besoins afin de budgéter leurs coûts. Cependant, prendre des décisions concernant le provisionnement à la volée des ressources et l'ordonnancement des calculs sur ces ressources est une tâche non-triviale, qui nécessite d'assister l'utilisateur dans sa prise de décision.

Notre thèse est que l'utilisation des *clouds* IaaS pour le calcul scientifique nécessite :

- **Un planificateur cloud** capable de gérer les applications et les ressources cloud pour l'utilisateur. Cet outil doit automatiser le provisionnement et la libération des ressources, ainsi que l'exécution des applications de l'utilisateur.

- **Un outil de prédiction** capable de fournir une estimation du coût de l'exécution d'une application. Cet outil doit prendre en compte les différentes stratégies de planification disponibles pour permettre à l'utilisateur de choisir la stratégie correspondant le mieux à ses contraintes.

Cette thèse propose plus spécifiquement des méthodes pour la construction d'outils de prédictions utilisant des simulateurs d'IaaS. Ce faisant, nous nous confrontons au problème de la variabilité des environnements de type cloud.

A.2 Contexte

A.2.1 Cloud computing

Un fournisseur IaaS opère généralement un ou plusieurs *data-centers*. Les machines physiques présentes dans le data-center sont partagées entre les clients sous forme de machines virtuelles, VMs. Chaque VM est affectée de manière opaque pour le client, à une, ou à un sous-ensemble de machines physiques. Différentes configurations permettent aux clients de choisir des VMs correspondant à leurs besoins. Les clients reçoivent les droits complets d'administration des VM louées. La location de VMs est facturée par intervalles atomiques de temps, qu'on appellera BTU, pouvant aller de la minute au mois selon les fournisseurs.

Le succès de l'IaaS s'explique par deux avantages majeurs en comparaison des infrastructures traditionnelles. D'une part, aucun investissement matériel n'est nécessaire de la part du client, qui connait à l'avance le coût intégral d'exploitation des ressources. D'autre part, la capacité de dimensionner à la volée la taille de l'infrastructure nécessaire permet une meilleure *consolidation* des machines physiques, c'est-à-dire permet de concentrer les calculs pour améliorer le taux d'utilisation des machines physiques. Ceci impacte positivement les coûts d'exploitation et les délais de disponibilité des machines pour les utilisateurs. La capacité à obtenir instantanément de nouvelles VMs permet à son tour aux utilisateurs de planifier des infrastructures plus légères ne provisionnant des ressources additionnelles que lorsque cela est nécessaire.

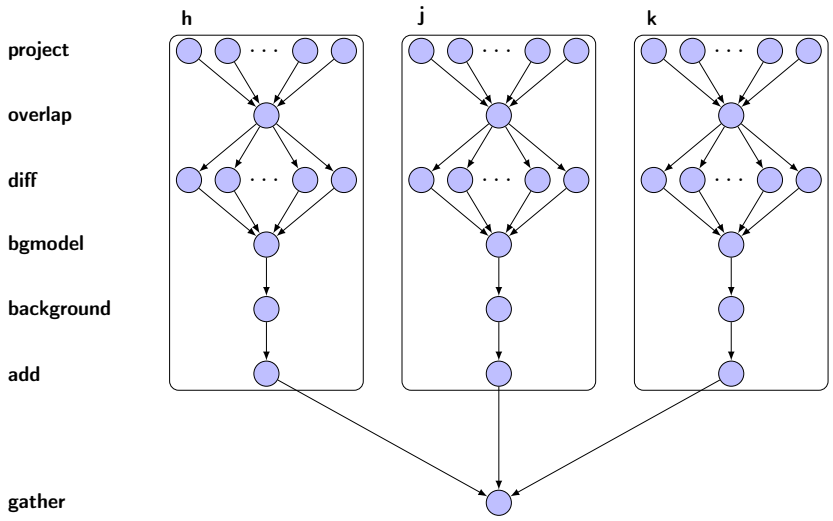

FIGURE A.1 – Flot d'exécution de Montage.

A.2.2 Calcul scientifique

La capacité des ordinateurs à gérer de grandes quantités de données a grandement étendu le champs des possibles dans la recherche scientifique. On s'intéressera spécifiquement aux applications destinées aux traitements par lots. Ces applications sont conçues comme un ensemble de tâches à exécuter. Deux types d'applications nous intéressent.

- Les **bag-of-task** (BoT) qui ne présentent aucune dépendance entre les différentes tâches. Un BoT peut être parallélisé à souhait.
- Les **workflow** qui présentent des dépendances de données entre les différentes tâches. Ces dépendances contraignent l'ordre d'exécution de certaines tâches. Les workflows peuvent être représentés sous forme de DAGs.

Ces applications, souvent conçues à l'origine pour être exécutées sur des grilles de calcul hétérogènes, sont capables de tourner sur une large variété d'équipements. Le contrôle de l'exécution de ces applications est généralement la responsabilité d'un programme planificateur externe à l'application. Hormis les dépendances présentes dans le cas d'un workflow, le planificateur se voit accorder une grande latitude sur la disposition des tâches sur les ressources disponibles. Ces types d'application de calcul scientifique sont des candidates de choix pour l'exécution dans le cloud.

Au cours de nos travaux nous nous sommes intéressé à l'exécution de deux applications de calcul scientifique qui servent de base à nos expériences. :

- **OMSSA** [21], un BoT utilisé en protéomique pour l'analyse des mesures d'un spectromètre de masse.
- **Montage** [22], un workflow utilisé en astronomie pour composer des images du ciel à partir de photographies prises à différents endroits et différents moments. Le workflow de Montage est représenté Figure A.1.

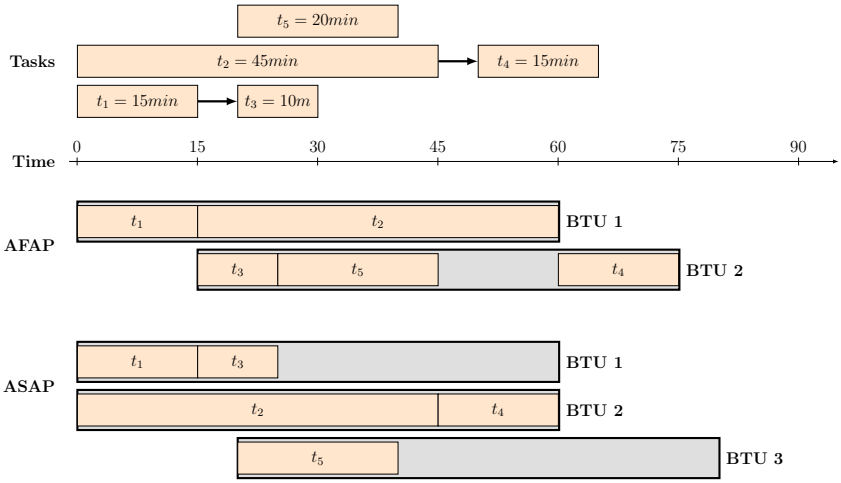

FIGURE A.2 – Exemple de planifications générées par AFAP et ASAP.

A.3 Calcul Scientifique dans le Cloud

A.3.1 Un planificateur : Schlouder

Schlouder [35] est un planificateur conçu dans notre équipe pour exécuter des applications de calcul scientifique de types BoT et workflow dans des clouds IaaS. En plus de gérer l'exécution des applications, Schlouder s'occupe du provisionnement des VMs et de l'ordonnancement des tâches sur les VMs provisionnées. Cette double gestion engendre un niveau de complexité supplémentaire en comparaison des ordonnanceurs pour grille qui calculent un ordonnancement pour un ensemble fixe de ressources. Schlouder propose des heuristiques pour optimiser deux objectifs : le temps total d'exécution de l'application, appelé *makespan*, et le coût d'exécution de l'application, lié aux décisions de provisionnement.

Dans nos travaux nous nous intéressons à deux heuristiques, représentées Figure A.2 :

- **AFAP**, *as full as possible*, tente d'optimiser le coût en remplissant systématiquement les VMs déjà provisionnées, si la BTU en cours n'est pas pleine.

- **ASAP**, *as soon as possible*, tente d'optimiser le makespan en réservant systématiquement de nouvelles VMs lorsque la tâche ne peut pas être exécutée immédiatement.

Les heuristiques ne donnent pas en général un ordonnancement optimal sur les deux objectifs. Les formes des workflows ou des contraintes extérieures, tels que les quotas d'utilisation, peuvent amener à produire des planifications dont l'objectif principal visé est en dessous des espérances de l'utilisateur. Devant la difficulté à anticiper les circonstances dans lesquelles une heuristique sera plus efficace qu'une autre, nous voulons proposer aux utilisateurs un outil permettant de prédire le temps et coût d'une exécution.

A.3.2 Un simulateur : SimSchlouder

Nous avons conçu SimSchlouder, un simulateur basé sur l'outil de simulation SimGrid, reproduisant le comportement d'une exécution de Schlouder. Nous pouvons ainsi proposer à l'utilisateur une prédiction du déroulement de l'exécution de l'application en ayant choisi l'une ou l'autre des heuristiques.

En comparant des exécutions réelles d'OMSSA et Montage faites avec Schlouder avec des exécutions simulées dans SimSchlouder, nous avons pu valider la justesse de SimSchlouder. Mais cette expérience a mis en lumière deux difficultés de l'utilisation de SimSchlouder comme outil de prédiction plutôt que de simulation. Premièrement, la quantité de données nécessaire pour obtenir une simulation précise est trop importante. Deuxièmement la variabilité, inhérente aux plateformes partagées comme les clouds, fait que plusieurs exécutions sur les mêmes données n'auront pas exactement le même comportement. Cette variabilité n'est pas prise en compte par SimSchlouder qui produit des simulations déterministes. Le résultat de la simulation est donc un résultat partiel de ce qui pourra être observé en réalité.

Enrichir le niveau d'information donné en simulation, prenant en compte la variabilité, nous conduit à créer un simulateur stochastique.

A.4 Contributions

Dans une simulation stochastique les entrées deviennent des variables aléatoires avec une distribution associée de valeurs possibles. Dans notre cas le temps d'exécutions des tâches, appelé *walltime*, est remplacé par un ensemble de wallitmes possibles. Le but de la simulation stochastique est de produire les distributions des makespans et des coûts d'exécutions possibles. Le passage à la simulation stochastique permet d'intégrer la variabilité observée dans nos exécutions réelles dans les simulations tout en diminuant la quantité et la précision des données nécessaires au simulateur. La prise en compte de la variabilité absorbe l'imprécision des données d'entrée.

A.4.1 Simulation de Monte-Carlo

La résolution de l'ordonnancement de DAG stochastiques via une méthode numérique s'est avérée inadaptée en raison des décisions dynamiques de placement que peut prendre Schlouder en cours d'exécution. Une alternative fréquemment choisie dans ce cas est le recours à la méthode de Monte-Carlo.

La *simulation de Monte-Carlo* (MCS), présentée en Figure A.3, permet de trouver les distributions de makespans et de coûts au travers de la répétition de simulations de scénarios possibles. La MCS tire pour chacune des tâches de l'application simulée un walltime en accord avec la distribution d'entrée. Cet ensemble de walltimes, appelé réalisation, représente un scénario d'exécution possible. Une réalisation contenant des walltimes fixés, elle peut donc être simulée avec SimSchlouder. Cette simulation nous

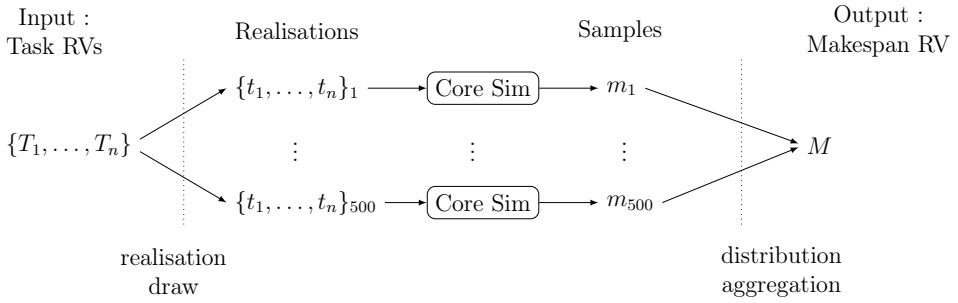

FIGURE A.3 – Principe d'une simulation de Monte-Carlo

donne un échantillon de makespans et de coûts possibles. En simulant suffisamment de réalisations différentes, nous obtenons suffisamment d'échantillons de makepans et de coûts pour estimer leur distribution.

La simulation de Monte-Carlo est extrêmement avantageuse : elle ne présente pas les contraintes mathématiques que possèdent les résolutions numériques, l'effet des heuristiques de planification est pris en compte par l'utilisation de SimSchlouder, et l'utilisation d'un simulateur déterministe au cœur de la MCS rend cette méthode facilement extensible et permet la validation indépendante du simulateur déterministe.

A.4.2 Distribution d'entrée

Il reste à choisir les distributions de walltimes utilisées comme entrée de la simulation. Notre objectif est de créer un outil de prédiction paramétrable de manière simple par son utilisateur. Nous sommes donc contraints dans le choix d'une distribution, car nous ne pouvons pas faire l'hypothèse que l'utilisateur possède un long historique d'exécutions passées de son application.

Dans les planificateurs de grilles comme dans Schlouder, il est demandé à l'utilisateur de fournir un temps d'exécution attendu pour chacune des tâches. Nous décidons de partir de cette donnée et d'ajouter un paramètre appelé niveau de perturbation (P). Le niveau de perturbation est identique pour toutes les tâches d'une simulation et représente la variabilité de l'exécution.

Concrètement, chaque tâche se voit attribuer une distribution uniforme $\mathcal{U}$ centrée sur le walltime attendu avec une largeur relative de $2P$. Ainsi avec $\bar{w_j}$ le walltime attendu pour la tâche j, sa distribution T_j sera :

$$T_j = \mathcal{U}(\bar{w_j}(1-P), \bar{w_j}(1+P))$$

Pour l'utilisateur cela revient intuitivement à considérer que la durée d'une tâche sera de $\bar{w_j} \pm P\%$. Ce modèle de distribution des walltimes est volontairement extrêmement simple, et il nous faut tester si une MCS avec ce modèle en entrée reste suffisamment précise pour nous fournir un encadrement juste des makespans et des coûts possibles de la réalité.

Application	Heuristique	Makespan		Coût
		CI 95%	CI 99%	
OMSSA	ASAP	90%	98%	100%
	AFAP	92%	100%	100%
Montage	ASAP	100%	100%	100%
	AFAP	100%	100%	100%

TABLE A.1 – Taux d'encadrement du makespan et du coût obtenus en utilisant les intervalles de confiance (CI) à 95% et à 99% de la distribution normale.

A.4.3 Validation expérimentale

Pour valider notre MCS utilisant nos distributions d'entrée nous comparons les résultats de notre MCS a un corpus d'exécutions réelles.

Pour isoler l'expérience des erreurs dans le choix des walltimes attendus, nous utilisons la valeur moyenne des walltimes observés pour chaque tâche. Le niveau de perturbation est lui aussi calculé à partir des observations réelles comme la moyenne des pires déviations vis-à-vis des walltimes attendus.

Les résultats de la MCS (Figure A.4) sont comparés quantitativement à la réalité par l'intermédiaire d'intervalles de confiance obtenus au travers de l'ajustement d'une distribution théorique sur l'ensemble des valeurs de makespan obtenues par MCS. Nous considérons que cette distribution est normale car le théorème de la limite centrale s'applique : le makespan est une somme de variables aléatoires (les walltimes), chaque variable aléatoire est indépendante, et nous avons suffisamment de tâches sur le chemin critique. Nous examinons ensuite pour différents intervalles de confiance de cette distribution, quelle part des observations réelles appartient à l'intervalle défini par l'intervalle de confiance choisi. Nous montrons par nos expériences qu'une MCS utilisant nos distributions d'entrée nous permet d'encadrer 90% des exécutions réelles. Les détails des résultats se trouvent dans la Table A.1.

A.5 Paramétrisation

Ayant obtenu un taux d'encadrement de 90% avec une distribution d'entrée simple, nous nous intéressons à l'effet sur la précision de la MCS du choix du niveau de perturbation (P), du nombre de réalisations (N), et du choix de la distribution d'entrée.

Le niveau de perturbation. Notre première intuition est que le niveau de perturbation (P) peut être augmenté pour produire des MCS ayant de meilleurs taux d'encadrement. Nos expériences avec des simulations nous montrent que ce n'est pas le cas. Certaines heuristiques de placement opèrent moins efficacement dans des environnements hautement variables. Il est donc nécessaire pour produire le meilleur encadrement possible que P soit le plus représentatif possible de la variabilité réelle de l'environnent simulé.

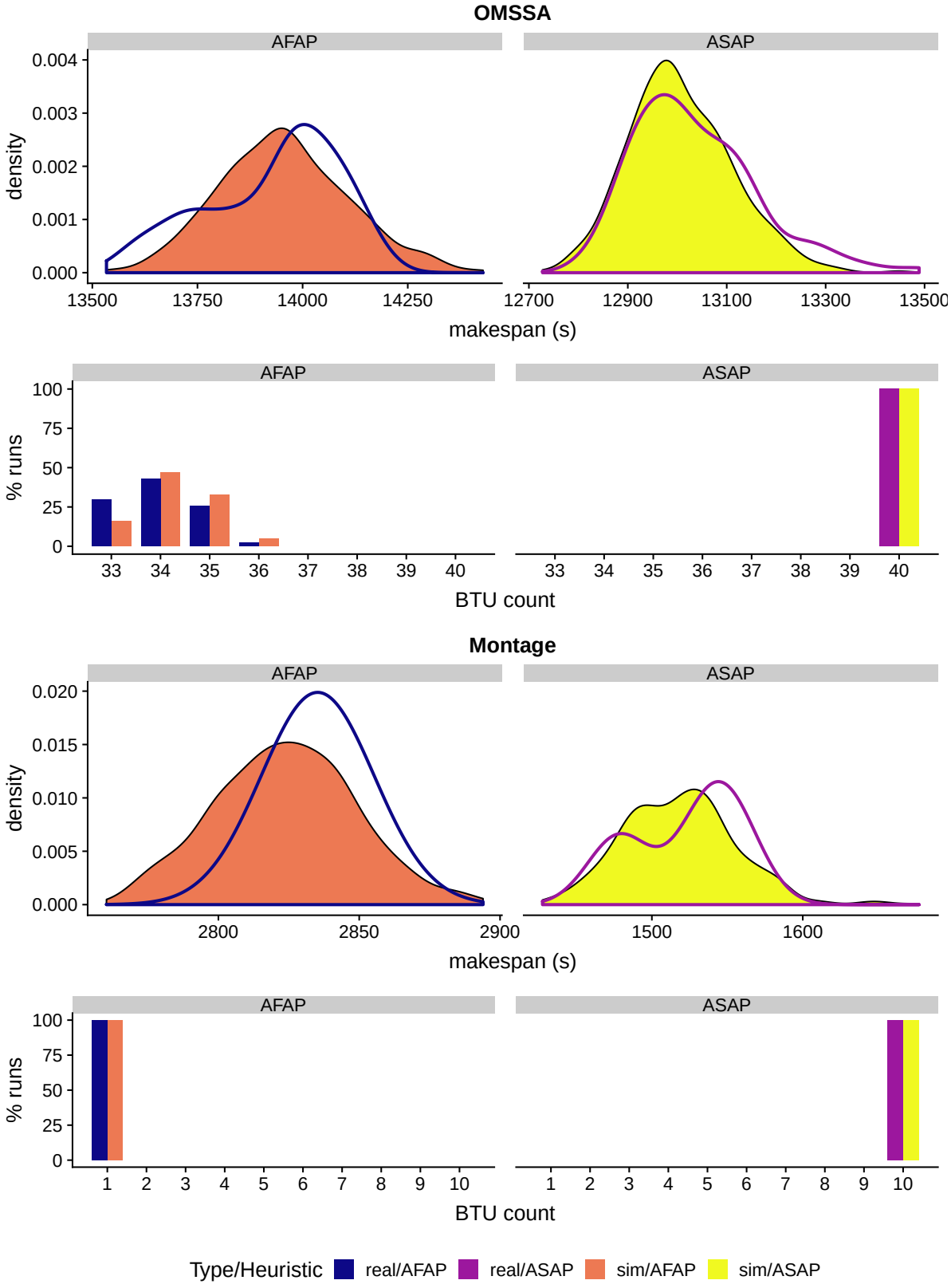

FIGURE A.4 – Comparaison des résultats de la MCS (en orange, jaune) avec les exécutions réelles (en bleu, violet).

Le nombre d'itérations. Les MCS sont des processus stochastiques. Il existe donc une variabilité dans les résultats d'une MCS d'une exécution à l'autre. De plus, le résultat évolue au fur est à mesure que des réalisations sont ajoutées à la MCS. La convergence des MCS ce fait en $1/\sqrt{N}$ où N est le nombre de réalisations utilisées dans la MCS. Pour des applications de la taille de OMSSA et Montage des résultats suffisants pour un encadrement du makespan et coût peuvent être obtenus avec seulement 500 réalisations.

La distribution d'entrée. Nous expérimentons aussi avec différentes distributions d'entrée. Ce changement n'a pas produit de meilleur taux d'encadrement ou une meilleure représentativité des exécutions réelles. Néanmoins l'utilisation de distributions plus complexes pourrait être avantageux dans les cas où l'utilisateur a accès aux données nécessaires à la construction de distributions plus proches des distributions observées dans la réalité.

A.6 Le cas Map-Reduce

Nos tentatives de simulation de Monte-Carlo dans le contexte de l'application Map-Reduce TeraSort démontrent certaines limites de notre approche :

- Notre distribution d'entrée utilisant un niveau de perturbation unique n'est pas adaptée pour représenter la variabilité d'environnements surchargés.
- Pour être efficace la MCS requiert qu'un nombre suffisant de tâches de taille similaire composent le chemin critique.

Dans notre simulation de TeraSort la tâche /reduce/ est beacoup plus longue que les tâches /map/. De plus le niveau de parallélisme observé limite la profondeur du chemin critique. Notre modèle MCS est donc inadapté à la simulation de cette application.

A.7 Conclusion

Les cloud présentent de nouvelles opportunités pour le calcul scientifique. Néanmoins, la complexité des systèmes de planifications nécessaires rend difficile la sélections d'une heuristique de planification pour l'utilisateur. Le développement des outils de prédiction susceptibles d'aider les utilisateurs dans cette sélection sont quant à eux confrontés à la quantité d'information nécessaire et à la variabilité observée sur les plateformes de cloud.

La thèse présentée ici est que les simulations stochastiques peuvent permettre, à partir d'un paramètrage simple pour l'utilisateur, de produire des prédictions pertinentes. Pour le démontrer nous avons développé une simulation de Monte-Carlo utilisant une distribution d'entrée simple. Cette simulation se montre capable d'encadrer 90% des exécutions réelles. Nous complétons ces travaux avec une étude détaillée de l'influence de différents paramètres de la simulation sur la précision des résultats.

Cette thèse présente une utilisation d'une simulation de Monte-Carlo dans un contexte de cloud, l'influence des différents paramètres sur le résultat et présente les limites de cette approche. Bien que nous soyons fier de ces résultats dans le contexte d'un outil de prédiction, nous voyons aussi les opportunités que les simulations stochastiques ouvrent en dehors de ce cas d'utilisation. Les MCS pourraient permettre l'étude précise du comportement des heuristiques de planification dans des environnements variables. En utilisant une simulation avec modèle de provisionnement stochastique, un opérateur de cloud pourrait développer et tester un algorithme de placement de VM maximisant l'efficacité énergétique tout en respectant ses obligations contractuelles de disponibilité. Les simulations de Monte-Carlo ouvrent la possibilité de créer des systèmes de cloud conscients de la variabilité de l'environnent dans lequel ils opèrent.

Appendix B

Publications

- Luke Bertot, Julien Gossa, and Stéphane Genaud. "Méthode pour l'étude expérimentale par la simulation de clouds avec SCHIaaS.". In: *Compas'17*. 2017
- Luke Bertot, Stéphane Genaud, and Julien Gossa. "An Overview of Cloud Simulation Enhancement using the Monte-Carlo Method". In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing CCGRID*. 2018
- Luke Bertot, Stéphane Genaud, and Julien Gossa. "Improving Cloud Simulation Using the Monte-Carlo Method." In: *Euro-Par 2018, LNCS 11014*. 2018

Université de Strasbourg

Luke Bertot
Improving the simulation of IaaS Clouds

msii ÉCOLE DOCTORALE

Résumé :

Les *clouds* sont devenus ces dernières années des plate-formes incontournables dans le monde informatique, car ils permettent de provisionner des ressources à la demande et de ne payer qu'à l'usage. Ceci ouvre la possibilité de concevoir de nouvelles stratégies pour la planification et l'exécution des applications parallèles de type tâches indépendantes ou workflow. Cependant, trouver une stratégie bien adaptée aux contraintes des utilisateurs, que ce soit en termes de coûts et de temps d'exécution, est un problème difficile, pour lequel des outils de prédictions sont nécessaires. Néanmoins, la variabilité inhérente de ces plate-formes complexifient le développement d'un tel outil de prédiction.
Notre thèse est que la simulation stochastique est une approche pertinente pour obtenir une prédiction s'accommodant de la variabilité, en produisant une distribution probabiliste des prédictions englobant les résultats réels observables. Pour le démontrer, nous utilisons une méthode de Monte-Carlo permettant de créer des simulations stochastiques par la répétitions de simulations déterministes. Nous montrons que cette méthode associée à certaines distributions d'entrée permettent de modéliser la variabilité d'une plate-forme à travers un unique paramètre. Pour évaluer la méthode proposée, nous comparons les résultats de notre méthode probabiliste à des exécutions réelles d'applications scientifiques. Nos expériences montrent que notre méthode permet de produire des prédictions représentatives des exécutions réelles observées.

Mots-clés : cloud computing, simulation de Monte-Carlo, calcul scientifique.

Abstract

The ability to provision resources on the fly and their pay-as-you-go nature has made *cloud computing* platforms a staple of modern computer infrastructure. Such platforms allow for new scheduling strategies for the execution of computing workloads. Finding a strategy that satisfies a user's cost and time constraints is a difficult problem that requires a prediction tool. However the inherent variability of these platforms makes building such a tool a complex endeavor.
Our thesis is that, by producing probability distributions of possible outcomes, stochastic simulation can be used to produce predictions that account for the variability. To demonstrate this we used Monte Carlo methods to produce a stochastic simulation by repeatedly running deterministic simulations. We show that this method used in conjunction with specific input models can model the variability of a platform using a single parameter. To validate our method we compare our results to real executions of scientific workloads. Our experiments show that our method produces predictions capable of representing the observed real executions.

Keywords: cloud computing, Monte Carlo simulations, scientific computing.