Excalibur: An Autonomic Cloud Architecture for Executing Parallel Applications

Alessandro Ferreira Leite, Claude Tadonki, Christine Eisenbeis, Tainá Raiol, Maria Emilia Walter, Alba Cristina Alves de Melo

Excalibur: An Autonomic Cloud Architecture for Executing Parallel Applications

Alessandro Ferreira Leite
Université Paris-Sud/University of Brasilia
alessandro.ferreira-leite@u-psud.fr

Claude Tadonki
MINES ParisTech / CRI
claude.tadonki@mines-paristech.fr

Christine Eisenbeis
INRIA Saclay / Université Paris-Sud
christine.eisenbeis@inria.fr

Tainá Raiol
Institute of Biology
University of Brasilia
tainaraiol@unb.br

Maria Emilia M. T. Walter
Department of Computer Science
University of Brasilia
mia@cic.unb.br

Alba Cristina Magalhães Alves de Melo
Department of Computer Science
University of Brasilia
albamm@cic.unb.br

Abstract

IaaS providers often allow the users to specify many requirements for their applications. However, users without advanced technical knowledge usually do not provide a good specification of the cloud environment, leading to low performance and/or high monetary cost. In this context, the users face the challenges of how to scale cloud-unaware applications without re-engineering them. Therefore, in this paper, we propose and evaluate a cloud architecture, namely Excalibur, to execute applications in the cloud. In our architecture, the users provide the applications and the architecture sets up the whole environment and adjusts it at runtime accordingly. We executed a genomics workflow in our architecture, which was deployed in Amazon EC2. The experiments show that the proposed architecture dynamically scales this cloud-unaware application up to 10 instances, reducing the execution time by 73% and the cost by 84% when compared to the execution in the configuration specified by the user.

Categories and Subject Descriptors C.2.4 [*Cloud computing*]: Software architecture

Keywords Cloud computing architecture, parallel execution, autonomic computing

CloudDP'14, April 13, 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2714-5/14/04. . . $15.00.
http://dx.doi.org/10.1145/2592784.2592786

1. Introduction

Nowadays, the cloud infrastructure may be used for high performance computing (HPC) purposes due to characteristics such as elastic resources, pay-as-you-go model, and full access to the underlying infrastructure [1]. These characteristics can be used to decrease the cost of ownership, to increase the capacity of dedicated infrastructure when it runs out of resources, and to respond effectively to changes in the demand. However, doing high-performance computing in the cloud faces some challenges such as differences in HPC cloud infrastructures and the lack of cloud-aware applications.

The cloud infrastructure requires a new level of robustness and flexibility from the applications, as hardware failures and performance variations become part of its normal operation. In addition, cloud resources' are optimized to reduce the cost to the cloud provider often without performance guarantees at low cost to the users. Furthermore, cloud providers offer different instance types (e.g. Virtual Machine (VM)) and services that have costs and performance defined according to their purpose usage. In this scenario, cloud users face many challenges. First, re-engineering the current applications to fit the cloud model requires expertise in both domains: cloud and high-performance computing, as well as a considerable time to accomplish it. Second, selecting the resources that fits their applications' needs requires data about the application characteristics and about the resources purpose usage. Therefore, deploying and executing an application in the cloud is still a complex task [14, 8].

Although some efforts have been made to reduce the cloud's complexity, most of them target software developers [12, 13] and are not straightforward for unexperienced users [8]. Therefore, in this paper, we propose and evalu-

ate an architecture to execute applications in the cloud with three main objectives: (a) provide a platform for high performance computing in the cloud for users without cloud skills; (b) dynamically scale the applications without user intervention; and (c) meet the users requirements such high performance at reduced cost.

The remainder of this paper is organized as follows. Section 2 presents our cloud architecture. In Section 3, experimental results are discussed. Section 4 presents related work and discusses cloud architectures to perform high-performance computing. Finally, Section 5 presents the conclusion and future work.

2. Design of the Proposed Architecture

Our architecture aims to simplify the use of the cloud and to run applications on it without requiring re-design of the applications.

We propose an architecture composed of micro-services. A micro-service is a lightweight and independent service that performs single functions and collaborates with other services using a well-defined interface to achieve some objectives. Micro-services make our architecture flexible and scalable since services can be changed dynamically according to users' objectives. In other words, if a service does not achieve a desirable performance in a given cloud provider, it can be deployed in another cloud provider without requiring service restart.

In this paper, an application represents a user's demand/-work, being seen as a single unit by the user. An application is composed of one or more tasks which represent the smallest work unit to be executed by the system. A partition is a set of independent tasks. The tasks that form an application can be connected by precedence relations, forming a workflow. A workflow is defined to be a set of activities, and these activities can be tasks, as said above, or even another workflows. The terms application and job are used interchangeably in this paper.

The proposed architecture has three layers: `Physical`, `Application`, and `User` layer (Figure 1). In the `Physical` layer, there are services responsible for managing the resources (e.g. Virtual Machine (VM) and/or storage). A resource may be registered by the cloud providers or by the users through the `Service Registry`. By default, the resources provided by the public clouds are registered with the following data: `resource type` (e.g. physical or virtual machine, storage), `resource URL`, `costs`, and resource `purpose` (e.g. if the resource is optimized for CPU, memory or I/O). The `Resource Management` service is responsible to validate these data and keep them updated. For instance, a resource registered at time t_i may not be available at time t_j, $t_j > t_i$, either because it failed or because its maximum allowed usage was reached. With the `Monitoring and Deployment` service, we deploy the users' jobs and monitor them. Monitoring is an important activity for many reasons. First, it collects data about the resources' usage. Second, it can be used to detect failures — sometimes the providers terminate the services when they are using/stressing the CPU, RAM memory, or both. And finally, to support the auto scaling mechanism.

We provide a uniform view of the the cloud providers' APIs implementing a `Communication API`. This is necessary because each provider may offer different interfaces to access the resources.

On top of the `Physical` layer, the `Application` layer provides micro-services to schedule jobs (Provisioning), to control the data flows (`Data Event Workflow`), to provide data streaming service (`Data Streaming Processing`), and to control the jobs execution. The architecture uses the MapReduce [3] strategy to distribute and execute the jobs. This does not mean that the users' applications must be composed of MapReduce jobs, but only that they are distributed following this strategy.

The `Coordination` service manages the resources, which can be distributed across different providers, and provides a uniform view of the system such as the available resources and the system's workload.

The `Provisioning` service uses high-level specifications to create a workflow. This workflow contains tasks which will set up the cloud environment and an execution plan which will be used to execute the application. In fact, the Provisioning service communicates with the `Coordination` service to obtain data about the resources and to allocate them for a job. After that, it submits the workflow for the `Workflow Management` service.

An execution plan consists of the application, the data sources, the resources to execute it, a state (`initializing`, `waiting data`, `ready`, `executing`, and `finished`), and a characteristic that can be `known` or `unknown` by the system. A characteristic represents the application's behavior such as CPU, memory, or I/O-bound.

The `Workflow Management` service coordinates the execution of the workflow and creates the data flows in the `Workflow Data Event` service.

The `Workflow Data Event` service is responsible for collecting and moving data for the execution plans. A data flow has a source and a sink and it can supply data for multiple execution plans. This avoids multiple accesses to the Distributed File System (DFS) to fetch the same data.

The `User` layer has two micro-services: `Job Submission` and `Job Stats Processing` services. A user submits a job using the `Job Submission` service. A job has the following data: the tasks which compose it, the constraints, the data definition (input and output), and the data about the cloud providers (name and access key).

The users can monitor or get the results of their jobs through the `Job Stats Processing`.

Scaling cloud-unaware application without technical skills requires an architecture that abstracts the whole environment, taking into account the users' objectives. In the

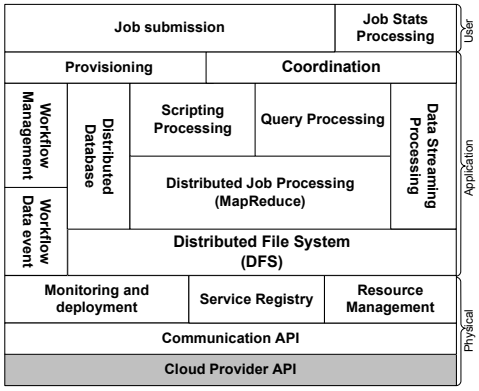


Figure 1. The proposed architecture and its micro-services.

next subsections, we explain how the proposed architecture achieves these goals.

2.1 Scaling cloud-unaware applications with budget restrictions and resource constraints

The applications considered in this paper are workflows but some parts of the workflow can be composed of a set of independent tasks that can be run in parallel. These independent tasks are the target of our scaling technique. They are split into P partitions, assigned to different resources. One important problem here is to determine the size and the number of partitions. Over-partitioning can lead to a great number of short duration tasks that may cause a considerable overhead to the system and can result in inefficient resources usage. To avoid this, a partition is estimated by [2]:

$$P = \frac{N_q * R}{T} \quad (1)$$

where N_q is the workload size; T is the estimated CPU time for executing N_q in the partition; and R is a parameter for the maximum execution time for partition P. A partition can be adjusted according to the node characteristics. For instance, if the resource usage by a partition P_i is below a threshold, P_i can be increased.

Partitions exist due to the concept of `splittable` and `static` files. It is the user who defines which data are splittable and how to split the data when the system does not know. Splittable data are converted to JavaScript Object Notation (`JSON`) records and persisted onto the distributed database, so a partition represents a set of `JSON` records. On the other hand, static data are kept in the local file system.

2.2 Minimizing data movement to reduce cost and execution time

Data movement can increase the total execution time of the application (makespan) and sometimes it can be higher than the computation time due to the differences in networks' bandwidth. In that case, we can invert the direction of the logical flow, moving the application as close as possible to the data location. Actually, we distribute the data using a Distributed File System and the MapReduce strategy.

Although MapReduce is an elegant solution, it has the overhead to create the `map` and the `reduce` tasks every time a query must be executed. We minimize this overhead using a data structure that keeps data in memory. This increases the memory usage and requires data consistency policies to keep the data updated, however it does not increase the monetary cost. We implemented a data policy that works as follows. For each record read by a node, it is kept in memory and its key is sent to the `Coordination` service (Figure 1). The `Coordination` stores the key/value pairs, where they were read. When a node updates a record, it removes the record from its memory and notifies the `Coordination` service. Then, the `Coordination` notifies asynchronously all the other nodes that have the key to remove it from memory.

2.3 Minimizing job makespan through workload adjustment

In an environment with incomplete information and unpredictable usage pattern as in the cloud, load imbalance can impact the total execution time and the monetary cost. For instance, assigning a CPU-bound task to a memory optimized node is not a good choice. To tackle this problem, we propose a workload adjustment technique that works as follows. For execution plans in the `ready` state and with an `unknown` application's characteristics, the scheduler selects similar execution plans and submits them for each available resource (i.e. CPU, memory or I/O optimized) and waits. As soon as the first execution finishes, the scheduler checks if there are similar execution plans in the `ready` state and submits them.

When there are no more `ready` execution plans, the scheduler assigns one in the `executing` state. Note that, in this case, the cost can increase, since we have more than one node executing the same task. In fact, we minimize this, finishing the slow node according to the difference between the elapsed time and the time to charge the node usage.

2.4 Making the cloud transparent for the users

As our architecture aims to make the cloud transparent for the users, it automates the setup process. However for some users, this is not sufficient since some jobs still require programming skills. For instance, consider the following scenarios: (i) a biologist who wants to search DNA units that have some properties in a genomics database, and to compare these DNA units with another sequence that he/she has built; (ii) a social media analyst who wants to filter tweets using some keywords.

Normally, these works require a program to read, parse, and filter the data. However, in our solution the users only have to know the data structure and to use a domain specific language (DSL) to perform their work. Listings 1 and 2 show how those works are defined, where `b`, `P1`, `P2`, `T` and `w` are users' parameters.

```
execute T with (select reads from genomic-
    database where P1 = X and P2 = Y) -seq = b
```

Listing 1. Specification of a genomics analysis application.

```
select tweet from tweets where text contains (w)
```

Listing 2. Specification of a Twitter analysis application.

In this case, a data structure (e.g. a file) is seen as a `table` whose fields can be filtered. Although we have similar approaches in the literature such as BioPig [12] and SeqPig [13], they still require programming skills to register the drivers and to load/store the data. In other words, to use them, the users have to know the system's internals.

In order to illustrate our architecture, consider the bioinformatics scenario described above. In this case, the biologist submits a XML or a YAML file with the application, the requirements, and the data definition (the genomics database and the built sequence) using a console application (a client of the `Job Submission` service) at the `User` layer. The `Job Submission` sends the job description to the `Provisioning` service at the `Application` layer and waits for the job's ID. When the `Provisioning` service receives the application, it executes the following steps. First it creates a workflow with five activities: (i) select the cheapest Virtual Machine to setup the environment; (ii) get non splittable files (e.g. a reference genome) to store them in the local file system; (iii) get the splittable files (the genomics database) and persist them into the DFS; (iv) create a Virtual Machine Image (VMI) of the configured environment; and (v) finish the VM used to configure the environment. Second, it selects the resources returned by the `Coordination` service that match the users' requirements or the applications' characteristics. Third, it creates an execution plan for the application; selects a resource to execute it; and starts the execution. Finally, it returns the job's ID.

In this scenario, a partition has a set of genomics sequences read from the DFS by the `Workflow Data Event` assigned to an execution plan. During the application's execution, the `Provisioning` service monitors the applications through the `Monitoring` service and if the partition's execution time reaches the expected time it creates more VMs to redistribute the workload. After all tasks have finished, the user receives the output through the `Job submission` service.

3. Experimental Results

We deployed an instance of our architecture on Amazon EC2. Our goal was to evaluate the architecture when instanced by a user without cloud skills.

We executed a genomics workflow that aims to identify non-coding RNA (ncRNA) in the fungi `Schizosaccharomyces pombe` (`S. pombe`). This workflow, called Infernal-Segemehl, consists of four phases (Figure 2): (i) first the tool Infernal[11] maps the `S. pombe` sequences onto a nucleic acid sequence database (e.g. Rfam [4]); (ii) then, the sequences with no hit or with a hit with a low score are processed by the segemehl tool[6] (iii) SAMTools[10] is used to sort the alignments and convert them to the SAM/BAM format. (iv) finally, the RNAFold tool[5] is used to calculate the minimum free energy of the RNA molecules obtained in step (iii).

Instance type	CPU	RAM	Cost ($/hour)
PC	Intel Core 2 Quad CPU 2.40GHz	4 GB	Not applicable
hs1.8xlarge	Intel Xeon 2.0 GHz 16 cores	171 GB	4.60
m1.xlarge	Intel Xeon 2.0 GHz 4 cores	15 GB	0.48
c1.xlarge	Intel Xeon 2.0 GHz 8 cores	7 GB	0.58
t1.micro	Intel Xeon 2.0 GHz 1 core	613 MB	0.02

Table 1. Resources used during the experiments.

We used the Rfam version 11.1 (with 2278 ncRNA families) and `S. pombe` sequences extracted from the EMBL-EBI (1 million reads). Rfam is a database of non-coding RNA families with a seed alignment for each family and a covariance model profile built on this seed to identify additional members of a family [4].

Although, in its higher level, this workflow executes only four tools, it is data oriented. In other words, each step processes a huge amount of data and, in all tools, each pairwise sequence comparison is independent. So, the data can be split and processed by parallel tasks.

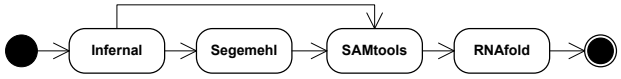


Figure 2. The Infernal-Segemehl workflow.

The Amazon EC2 micro instance (t1.micro) was used to setup the environment (e.g. install the applications, copy the static files to the local file system) and to create a Virtual Machine Image (VMI). We chose it because it is cheaper and also eligible for the free quota.

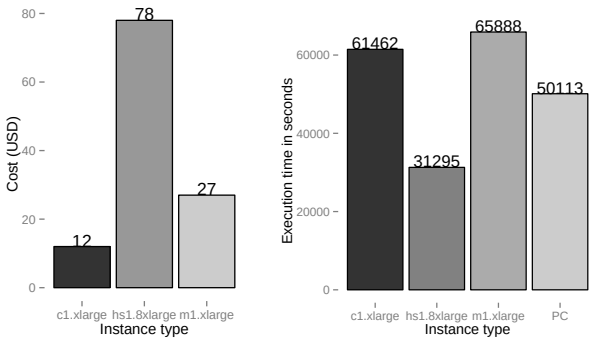
In addition to the cloud's executions, we also executed the workflow in a local PC (Table 1) to have an idea of the cloud overhead.

3.1 Case study 1: execution without auto scaling

This experiment aims to simulate the users' preferences, where an instance is selected either upon their knowledge about the applications' requirements or the amount of computational resources offered by an instance. We executed the workflow in the first four instances listed in Table 1. The t1.micro instance was used exclusively to setup the environment and it was not used to run the application.

Figure 3 shows the costs and execution time for the four instances. The time was measured from the moment the application was submitted until the time all the results are produced (wallclock time). Therefore, it includes the cloud overhead (data movement to/from the cloud, VM instantiation, among others). The instance `hs1.8xlarge`, which was selected based on the application requirements ($\geq$ 88GB of RAM), outperformed all other instances. Although it was possible for the user to execute his/her application without any technical cloud skills, the amount paid (USD 78.00) was high. This happened because the user specified that the application would need more than 88GB of RAM and in fact, the application used only 3GB of RAM.

Considering this scenario, the cloud is not an attractive alternative for the users due to its execution times; those were 22% and 31% higher than the local execution (PC Table 1). Even in the best configuration (`hs1.8xlarge`), the execution time was only 60% lower with a high monetary cost. These differences are owing to the multitenant model employed by the clouds.



(a) Cost (b) Execution time

Figure 3. Cost and execution time of the Infernal-Segemehl workflow (Figure 2) in the cloud allocating the resources based on users' preferences.

3.2 Case study 2: execution with auto scaling

This experiment aims to evaluate if the architecture can scale a cloud-unaware application.

Based upon the previous experiment (Figure 3), the system discarded the I/O optimized instance (`hs1.8xlarge`) due to its high cost(Table 1) and also because the application did not really require the amount of memory defined by the user. In a normal scenario, this instance is selected only if the monitoring service confirms that the application is I/O intensive.

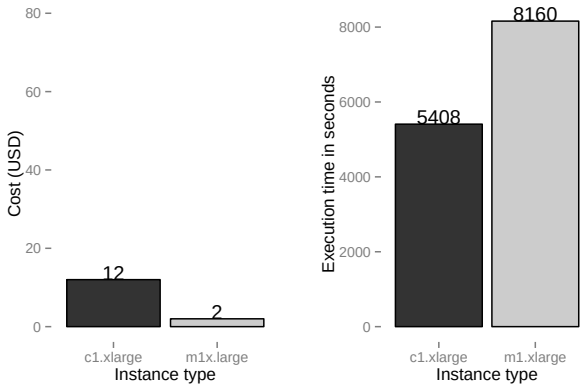
To scale, the system creates P partitions $(P_1, P_2, \cdots, P_n)$ using the Equation 1 (Section 2.1) with R equals to 1 hour and T equals to 9 hours. These values represent respectively the expected execution time for one partition P_i and for the whole workflow. They were defined because Amazon charges the resource by the hour and because, in the previous experiment, the best execution time took approximately 9 hours to finish (Figure 3). This means that this experiment aims to at least decrease the cost. In this case, were created 9 partitions.

As the beginning, the system had not sufficient data to decide if the workflow was memory or CPU-bound, so it submitted two similar partitions — for two instance types (`m1.xlarge` and `c1.xlarge`) — to realize which was the most appropriate for the partition.

Figure 4 shows the execution time for each partition in the selected instance types. As soon as execution in the partition assigned to the `c1.xlarge` instance finished, the system created one VM for each partition in the `ready` state and executed them. Although there were only 7 partitions in the ready state and 1 in execution (execution state) the architecture duplicates the partition in execution, since its execution time in the `m1.xlarge` instance was unknown. After one hour, more three instances were created to redistribute the tasks as shown in Figure 5.

Due to the cloud infrastructure, which provided in nearly real time the requested resources, and the auto scaling mechanism, which selected the resources based on the partitions' characteristics, we decreased both the cost (5 times) and the makespan (10,830 seconds) using 10 `c1.xlarge` instances (80 vCPUs) and one `m1.xlarge` (4 vCPUs).

Our strategy differs from the scaling services[1] offered by the cloud providers, since the users do not have to select an instance type nor to split the work manually.



(a) Cost to execute the workflow using 10 c1.xlarge instances and 1 m1.xlarge instance.

(b) Execution time for one partition when executed in the c1.xlarge and m1.xlarge instances. One partition was defined to finish in 1 hour with the deadline of 9 hours for the workflow.

Figure 4. Cost to execute the workflow (Figure 2) with auto scaling enabled.

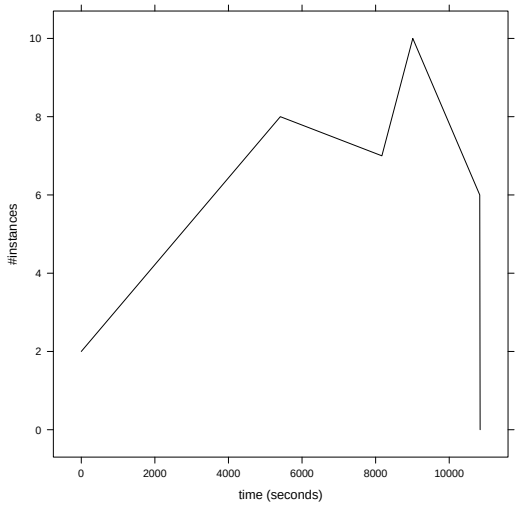


Figure 5. Scaling the Infernal-Segemehl workflow.

[1] Amazon CloudWatch (`aws.amazon.com/cloudwatch/`).

4. Related Work

In the last years, many works have described the challenges and opportunities of running high-performance computing in the cloud[1, 8]. Many of the benefits identified by these works, such as easy access to the resources, elasticity, stability and resource provisioning in nearly real time, as well as the technical skills required to administrate the cloud, confirms the requirements of reducing the complexity for the users, and are consistent with our work in scaling the workflow Infernal-Segemehl using the Amazon EC2.

Recently, many works have focused on developed new architecture to execute users' applications in the cloud considering both cost and performance. For instance, the Cloud Virtual Service (CloVR)[2] is a desktop application for automated sequences analyses using cloud computing resources. With CloVR, the users execute a VM on the their computer, configure the applications, insert the data in a special directory, and CloVR deploys an instance of this VM on the cloud to scale and to execute the applications. CloVR scales the application by splitting the workload in P partitions using Equation 1, and the Cunningham BLAST runtime to estimate the CPU time for each BLAST query.

In [9], biological applications are run on the Microsoft Windows Azure showing the required skills and challenges to accomplish the work. Iordache and colleagues[7] developed Resilin, an architecture to scale MapReduce jobs in the cloud. The solution has different services to provision the resources, to handle jobs flow execution, to process the users requests, and to scale according to the load of the system. Doing bioinformatics data analysis with Hadoop requires knowledge about the Hadoop internal and considerable effort to implement the data flow. In [12], a tool for bioinformatics data analysis called BioPig is presented. In this case, the users select and register a driver — bioformatics algorithms — provided by the tool and write their analysis' jobs using the Apache Pig (`pig.apache.org`) data flow language. SeqPig [13] is another tool that has the same objective of BioPig. The differences between them are the drivers provided by each tool. These tools reduce the needs to know Hadoop internal to realize bioinformatics data analysis.

The closest works to ours are [2], [12], and [13]. Our work differs from these approaches in the following ways. First, the users do not need to configure a VM in their computers to execute the applications in the cloud. Second, our architecture tries to match the workload to the appropriate instance type. Third, the data flow is defined using an abstract language freeing the users to write any code. The language is the same as used by BioPig and SeqPig but with the difference that the users write the data flow only considering the data structure. For instance, to filter the sequences using BioPig or SeqPig the users have to register the loaders, the drivers, and write a script to execute the analysis, which is more appropriate for software developers.

5. Conclusion and Future Work

In this paper, we proposed and evaluated a cloud architecture based on micro-services to execute application in the cloud.

With a user-oriented perspective, we could execute a genomics workflow without requiring programming skills or cloud knowledge from the users. We executed two experiments using Amazon EC2 to evaluate the architecture when instantiated with and without auto scaling. In the first case, the user was responsible to define an instance type to execute the workflow without auto scaling. In the second case, an instance type was selected based on the applications' characteristics and the work was split to reduce the execution time. Using 11 VMs we decreased both the cost and the execution time when compared to an execution without the auto scaling.

As future work, we intend to instantiate the architecture running other applications in a hybrid cloud. Also, we will consider a dynamic scenario, where both the number of tasks are unknown and the resources usage are restricted. Finally, we intend to incorporate QoS and budget requirements.

Acknowledgments

The authors would like to thank CAPES/Brazil and CNPq/Brazil though the STIC-AmSud project BioCloud, and INRIA/France for their financial support.

References

[1] M. AbdelBaky et al. "Enabling High-Performance Computing as a Service". In: *Computer* 45.10 (2012), pp. 72–80.

[2] S. Angiuoli et al. "CloVR: A virtual machine for automated and portable sequence analysis from the desktop using cloud computing". In: *BMC Bioinformatics* 12.1 (2011), pp. 1–15.

[3] J. Dean et al. "MapReduce: simplified data processing on large clusters". In: *6th OSDI*. Vol. 6. USENIX, 2004.

[4] S. Griffiths-Jones et al. "Rfam: annotating non-coding RNAs in complete genomes." In: *Nucleic Acids Research* 33 (1 2005), pp. D121–D124.

[5] I. Hofacker et al. "Fast folding and comparison of RNA secondary structures". In: *Chemical Monthly* 125.2 (1994), pp. 167–188.

[6] S. Hoffmann et al. "Fast Mapping of Short Sequences with Mismatches, Insertions and Deletions Using Index Structures". In: *PLoS computational biology* 5 (9 2009), e1000502.

[7] A. Iordache et al. "Resilin: Elastic MapReduce over Multiple Clouds". In: *13th IEEE/ACM CCGrid* 0 (2013), pp. 261–268.

[8] G. Juve et al. "Comparing FutureGrid, Amazon EC2, and Open Science Grid for Scientific Workflows". In: *Computing in Sci* 15.4 (2013), pp. 20–29.

[9] J. Karlsson et al. "Enabling Large-Scale Bioinformatics Data Analysis with Cloud Computing". In: *10th IEEE ISPA*. 2012.

[10] H. Li et al. "The Sequence Alignment/Map format and SAMtools". In: *Bioinformatics* 25.16 (Aug. 2009), pp. 2078–2079.

[11] E. P. Nawrocki et al. "Infernal 1.0: inference of RNA alignments". In: *Bioinformatics* 25.10 (2009), pp. 1335–1337.

[12] H. Nordberg et al. "BioPig: a Hadoop-based analytic toolkit for large-scale sequence data". In: *Bioinformatics* 29.23 (2013), pp. 3014–3019.

[13] A. Schumacher et al. "SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop". In: *Bioinformatics* 30.1 (2014), pp. 119–120.

[14] Y. Zhao et al. "Opportunities and Challenges in Running Scientific Workflows on the Cloud". In: *CyberC*. Oct. 2011, pp. 455–462.