Journal of Physics: Conference Series **649** (2015) 012006 doi:10.1088/1742-6596/649/1/012006

Towards Cloud-based Asynchronous Elasticity for Iterative HPC Applications

Rodrigo da Rosa Righi[1], Vinicius Facco Rodrigues[1], Cristiano André da Costa[1], Diego Kreutz[2] and Hans-Ulrich Heiss[3]

[1] Applied Computing Graduate Program - Unisinos - Av. Unisinos, 950 – São Leopoldo, RS, Brazil
[2] SnT, University of Luxembourg - 4, rue Alphonse Weicker L-2721 Luxembourg
[3] Technische Universität Berlin - Sekretariat EN 6, Einsteinufer 17 D-10587 Berlin

E-mail: rrrighi@unisinos.br, viniciusfacco@live.com, cac@unisinos.br, diego.kreutz@uni.lu, hans-ulrich.heiss@tu-berlin.de

Abstract. Elasticity is one of the key features of cloud computing. It allows applications to dynamically scale computing and storage resources, avoiding over- and under-provisioning. In high performance computing (HPC), initiatives are normally modeled to handle bag-of-tasks or key-value applications through a load balancer and a loosely-coupled set of virtual machine (VM) instances. In the joint-field of Message Passing Interface (MPI) and tightly-coupled HPC applications, we observe the need of rewriting source codes, previous knowledge of the application and/or stop-reconfigure-and-go approaches to address cloud elasticity. Besides, there are problems related to how profit this new feature in the HPC scope, since in MPI 2.0 applications the programmers need to handle communicators by themselves, and a sudden consolidation of a VM, together with a process, can compromise the entire execution. To address these issues, we propose a PaaS-based elasticity model, named AutoElastic. It acts as a middleware that allows iterative HPC applications to take advantage of dynamic resource provisioning of cloud infrastructures without any major modification. AutoElastic provides a new concept denoted here as asynchronous elasticity, i.e., it provides a framework to allow applications to either increase or decrease their computing resources without blocking the current execution. The feasibility of AutoElastic is demonstrated through a prototype that runs a CPU-bound numerical integration application on top of the OpenNebula middleware. The results showed the saving of about 3 min at each scaling out operations, emphasizing the contribution of the new concept on contexts where seconds are precious.

1. Introduction

One of the key features of the cloud includes the elasticity, where users can scale at any moment their resource consumption up or down according to either the demand or the desired response time [1, 2]. Considering the HPC landscape and a very long running parallel application, a user may want to increase the number of instances to try and reduce the completion time of the application. On the other hand, if an application is not scaling in a linear or close to linear way, and if the user is flexible with respect to the completion time, the number of instances can be reduced. This results in a lower nodes × hours index, and thus in a lower cost and energy saving. Although there are benefits to HPC systems, cloud elasticity has been more extensively explored on client-server Web architectures, such as video on demand, online stores,

BOINC applications, e-governance and Web services [2]. As illustrated in Figure 1, a typical strategy in this context uses horizontal cloud elasticity to replicate virtual machine instances in a datacenter [3, 4]. Despite transparent to the user, this kind of mechanism is suitable on loosely coupled programs in which replicas do not establish communication among themselves [5, 6].

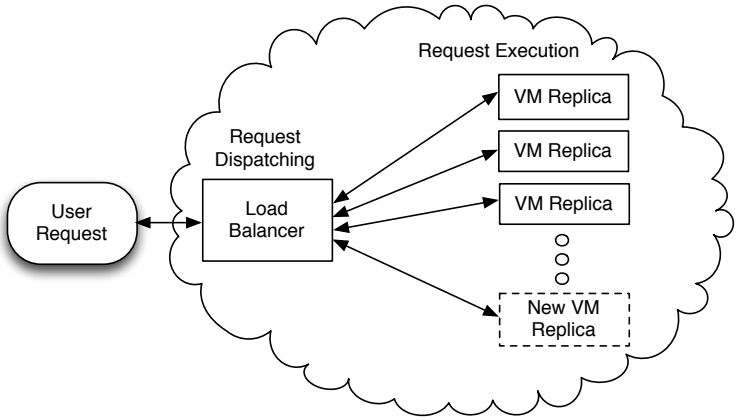


Figure 1. Standard cloud elasticity mechanism: horizontal elasticity and an elasticity controller acting as load balancer.

Although pertinent for bag-of-tasks and key-store HPC applications, replication techniques and centralized load balancers are not useful by default to implement elasticity on tightly-coupled HPC applications, as those modeled as Bulk-Synchronous Parallel (BSP), Divide-and-Conquer or pipeline [2, 7]. This happens because any resource (de)allocation causes a process reorganization as well as the updating of the whole communication topology, not only the interaction between the load balancer and the target replicas. In addition, there is a problem related to virtual machine consolidation, which can result in a sudden termination of a process and its disconnection from the communication topology; and consequently, resulting in the application crash. Most parallel applications have been developed using the MPI 1.x, which means that they do not have any support for changing the number of processes during the execution, so applications cannot explore elasticity without an appropriate support [8]. While this changed with MPI version 2.0, significant effort is needed at application level both to manually change the process group and to redistribute the data to effectively use a different number of processes.

Figure 2 (a) depicts a situation in which elasticity controls are implemented inside the application code using the cloud-supported API. This strategy requires user expertise on cloud monitoring, besides the selection of the appropriate points to insert the calls. Part (b) of Figure 2 explores the use of an elasticity controller outside the application, which is normally offered as optional component in platforms such as Amazon and Windows Azure [9]. Resource monitoring, as well as allocation and deallocation of VMs are tasks belonging to the controller, but users must both insert calls in their applications and handle the communication topology reorganization. The call of the *elasticity()* method represents a link between the application and the controller, so the use of a controller without it has no effect in load balancing because of the application is not able to detect and use new resources [10]. To bypass these limitations, some approaches impose code rewriting [2, 11], previous configuration of elastic rules and actions [2, 12, 13, 14], former knowledge of the application phases [2, 12, 13, 14], and the stop-reconfigure-and-go [2] mechanism, to obtain gains from resource reconfiguration.

Aiming at providing cloud elasticity for HPC applications in an efficient and transparent

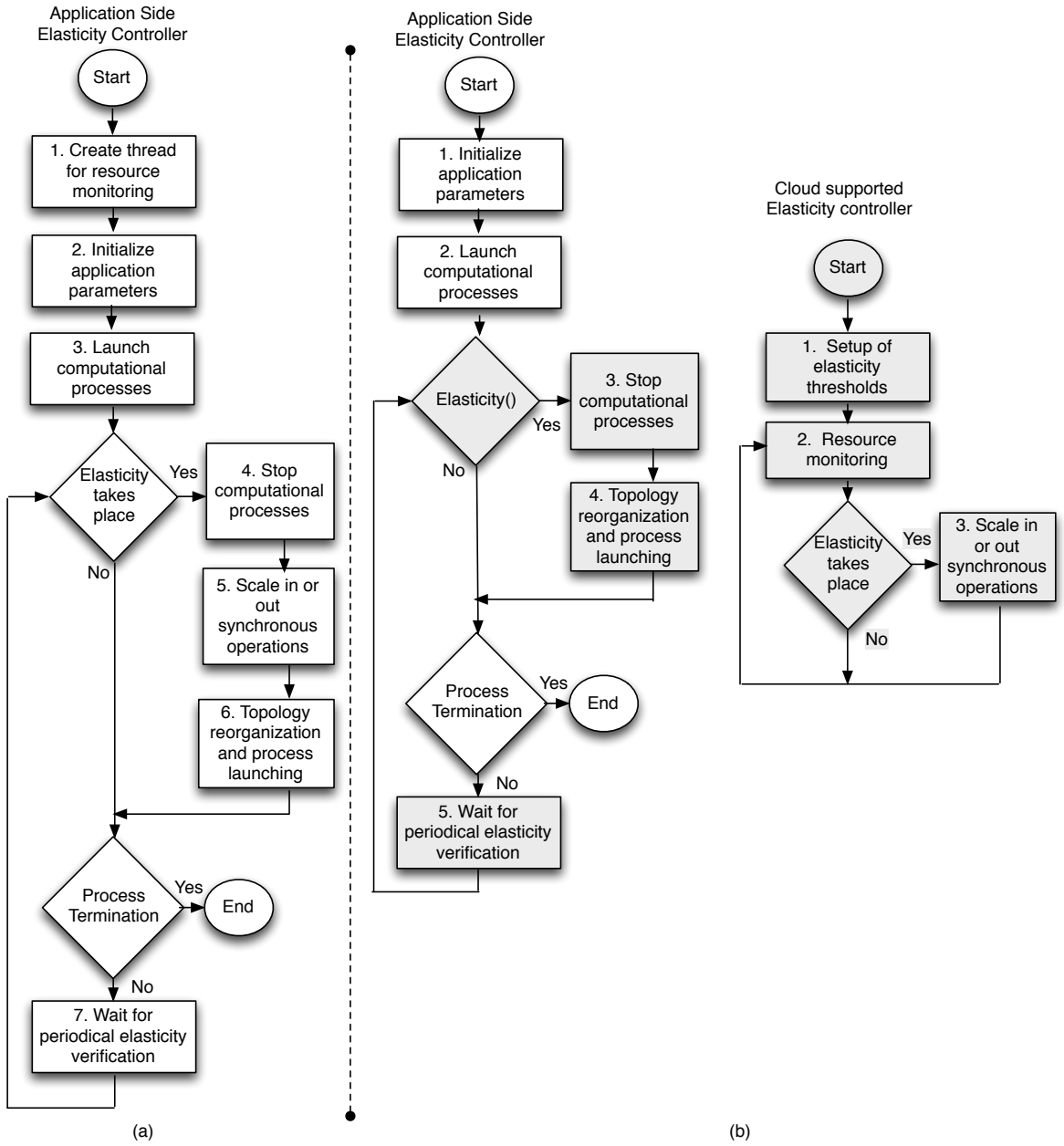


Figure 2. Different approaches for cloud elasticity: (a) elasticity actions are managed directly at application code; (b) use of an elasticity controller outside the application, which can offer a concomitance between elasticity and application's processes actions. However, this approach is not transparent to the users, who need to test elasticity actions and reorganize the communication topology by themselves. On the other hand, AutoElastic offers the concept of asynchronous elasticity by offering a framework to address the aforementioned asynchronism totally transparent from the user. To accomplish this, AutoElastic addresses the shading boxes of part (b) which are put apart from the user responsibility.

manner, we are proposing a PaaS-based model called **AutoElastic**[1]. Particularly, AutoElastic is focusing on master-slave iterative applications, but offers a flexible framework to support

[1] Project website: http://autoelastic.com

other HPC programming styles, such as pipeline and BSP. AutoElastic's contribution relies on the concept of asynchronous elasticity: transparent resource and process reorganization at user perspective, neither blocking nor stopping the application execution at any resource allocation or deallocation action. To accomplish this, AutoElastic provides a framework with a controller that transparently manages horizontal elasticity actions, *i.e.*, without requiring any application modification or adptation. Taking at starting point Figure 2 (b), our approach offers a framework to hide all shading boxes from the user. Although the standard use of a controller enables the setup of VMs in parallel to the application runtime, the benefits of the new resources are not transparent to the users. As discussed earlier, scaling in operations also appear as a problem in the standard utilization of a controller, since a consolidation of one or more VMs will sudden terminate the processes residing on them, which can imply in a premature application ending.

The proposed model assumes that the target HPC application is iterative by nature, *i.e.*, it has a time-step loop. This is a reasonable assumption for the most of MPI programs [15, 16], so this does not limit the applicability of our model. This article describes AutoElastic and a prototype developed with OpenNebula. Tests with a CPU-bound numeric integration application show gains up to 26% when using AutoElastic in comparison with a static provisioning. The remainder of this article will first introduce the related work in Section 2, pointing out open issues and research opportunities. Section 3 is the main part of the article, describing AutoElastic's framework together with asynchronous elasticity concept in details. Section 4 describes a prototype implementation. Evaluation methodology and results are discussed in Sections 5 and 6. Finally, Section 7 emphasizes the scientific contribution of the work and notes several challenges that we can address in the future.

2. Related Work

Elasticity is one of the most attractive features of cloud computing because it allows users to scale resources on-demand. There are different ways of using the elasticity provided by cloud infrastructures, such as manual setup [17, 18, 19], and by pre-configuration of reactive elastic mechanisms [20, 9]. While the former is not suitable for applications that need automatic and transparent elasticity, the latter implies in rather complicated tasks for non-cloud savvy users (e.g., define thresholds and elasticity actions).

Amazon AWS (`http://aws.amazon.com`), Nimbus (`http://www.nimbusproject.org`) and Windows Azure (`http://azure.microsoft.com`) are examples of systems that provide elasticity through pre-configured reactive mechanisms. Middleware solutions for building elastic computing infrastructures, such as OpenStack (`https://www.openstack.org`), OpenNebula (`http://opennebula.org`), Eucalyptus (`https://www.eucalyptus.com`) and CloudStack (`http://cloudstack.apache.org`), commonly offer elasticity through manual mechanisms (e.g., command line and graphical tool that allow users to control virtual machines). Complementary solutions such as Elastack [21], which provides automated monitoring and adaptation functions, can be integrated with OpenStack-like systems to provide dynamic infrastructure elasticity. However, it works only at the infrastructure level, *i.e.*, applications have to be made aware that nodes can be started or shut down at any time. In other words, it is up to the developers to ensure any kind of consistency or failure tolerance in the applications.

More recently, different research initiatives started to look at how elasticity can be leveraged by HPC applications. As an example, ElasticMPI proposes an elasticity framework for MPI applications through the stop-reconfigure-and-go approach [2]. However, this approach can negatively impact the performance of applications, in particular those that do not have long execution times. A second drawback of ElasticMPI is that it requires applications to be modified. Another approach, named Auto-elasticity [22], considers a pre-defined auto-elasticity by adjusting the number of VM instances accordingly to the application's input data (workload). In other words, as Auto-elasticity assumes that a program is modeled on deadline basis, the

number of VMs is pre-defined in order to meet the deadlines.

Most of the existing solutions that provide cloud elasticity for high performance applications are commonly built around the master-slave programming model [2, 11, 23]. In case of iterative applications, which is the most common one, it means that at each new loop the master redistributes the tasks to slaves [2, 11]. However, in most cases the elasticity of the system is provided in a reactive way at the IaaS level, *i.e.*, without knowledge of on-the-fly information from the applications. Summing up, current approaches suffer from different issues such as (i) lack of mechanism to verify whether the application reached (or not) its peak load when achieving a load balancing threshold value [21, 23]; (ii) extra complexity at the application level, *i.e.*, the code needs to be instrumented and/or reorganized [2, 11]; (iii) static elasticity defined by pre-execution information [2, 14]; (iv) reconfiguration of the application's resources using a stop-and-relaunch approach [2]; and (v) assume that the communication latency between any two VMs is constant [24].

Considering the scope of MPI applications, Raveendran, Bicer and Agrawal [2] proposed one of the most advanced approaches to support the execution of such kind of applications. Nevertheless, as mentioned above, their solution needs application data in advance to feed the elasticity middleware and the insertion of elasticity code in the MPI application, besides the need to stop and relauching the whole application when elasticity takes place. Observing the initiatives described here, we are prooposing AutoElastic as a first step towards addressing the aforementioned issues (i), (ii), (iii), and (iv). In other words, our solution does not add any extra code or complexity to the existing HPC applications, allows dynamic (runtime) elasticity, and enables on-the-fly reconfiguration of resources without having to stop and relaunch the application.

3. AutoElastic Model

Traditionally, HPC applications are executed on clusters or even in grid architectures. In general, both have a fixed number of resources that must be maintained in terms of infrastructure configuration, scheduling (where tools such as PBS[2], OAR[3], OGS[4] are usually employed for resource reservation and job scheduling) and energy consumption. In addition, the tuning of the number of processes to execute a HPC application can be a hard procedure: (i) both short and large values will not explore the distributed system in an efficient way; (ii) a fixed value cannot fit irregular applications, where the workload varies along the execution and/or sometimes it is not predictable in advance. On the other hand, cloud elasticity abstracts the infrastructure configuration and technical details about resource scheduling from users, who pay for resources, and energy consequently, in accordance with the application's demands. However, the main gaps between the duet HPC and elasticity are application modeling and the overhead related to scaling out operations. Aiming at addressing these gaps, we propose AutoElastic – a cloud elasticity model that operates at the PaaS level of a cloud, acting as a middleware that enables the transformation of a non-elastic parallel application in an elastic one. Thus, AutoElastic was proposed as a solution to answer questions such as:

(i) *Is it possible to provide cloud elasticity to high performance computing applications in a transparent and non-intrusive way (i.e., without needing to modify applications)?*

(ii) *Which HPC applications can benefit from cloud elasticity and what are the gains of using cloud elasticity?*

(iii) *What are the minimal assumptions to transparently support cloud elasticity in HPC applications?*

[2] Project Website: http://www.arc.ox.ac.uk/content/pbs

[3] Project Website: https://oar.imag.fr/

[4] Previously known as Sun Grid Engine (SGE). Project Website: http://gridscheduler.sourceforge.net

AutoElastic provides transparent horizontal and reactive elasticity for parallel applications, *i.e.*, without requiring the intervention of the programmer (also named here as cloud user) to specify sets of rules, actions, or modify the application's code. Figure 3 (a) illustrates the traditional approaches of providing cloud elasticity to HPC applications, while (b) highlights AutoElastic's idea. The approach proposed by AutoElastic allows users to submit a traditional, non-elastic aware, application to the cloud, while the framework takes care of resource reorganization through automatic VM allocation and consolidation procedures. As AutoElastic works at the granularity of virtual machines, it has to be aware of the VM instantiation overhead to provide seamless elasticity, *i.e.*, in a non-prohibitive way for HPC applications.

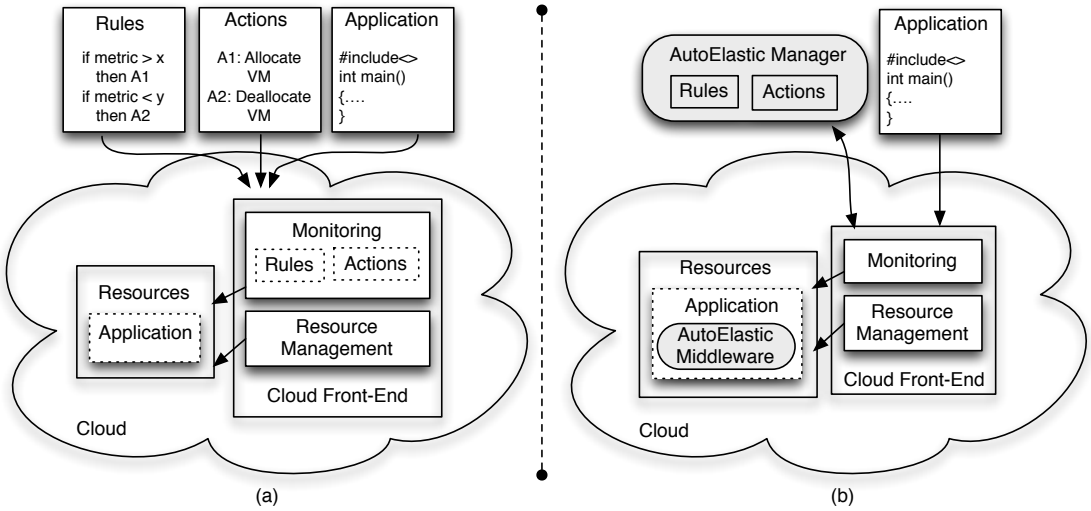


Figure 3. General ideas on using elasticity: (a) standard approach adopted by Amazon AWS and Windows Azure, in which the user must pre-configure a set of elasticity rules and actions; (b) AutoElastic idea, contemplating a manager that coordinates the elasticity actions and configurations on behalf of the user.

3.1. Architecture

AutoElastic is a middleware that operates as PaaS (Platform as a Service) that allows non-elastic parallel applications to take advantage of cloud elasticity without any change. To provide elasticity, it works with scaling in and out operations that consolidate or allocate virtual machine instances, respectively. Figure 4 depicts the AutoElastic architecture, presenting the framework components and the mapping of VMs. The framework includes a Manager, which can be either assigned to a virtual machine inside the cloud or to act as a stand-alone program outside the cloud. This is possible by taking advantage of cloud-supported APIs. As HPC applications are commonly CPU-bounded, we opted to create a process per VM and c working VMs per computing node, where c refers to the number of computational cores inside the node. This design decision has been previously investigated and validated as a way of exploring the efficiency of large computing nodes [25]. In addition, Figure 4 also presents the first ideas regarding the scope of HPC applications, presenting VMs that execute master and slave processes.

The AutoElastic Manager monitors the virtual machines, taking elasticity actions when considering them as pertinent for the current hardware and application behavior. The user can inform a file with an SLA (Service-Level Agreement) containing the minimum and the maximum number of allowed VMs to execute the application on the cloud. If no SLA is provided, the

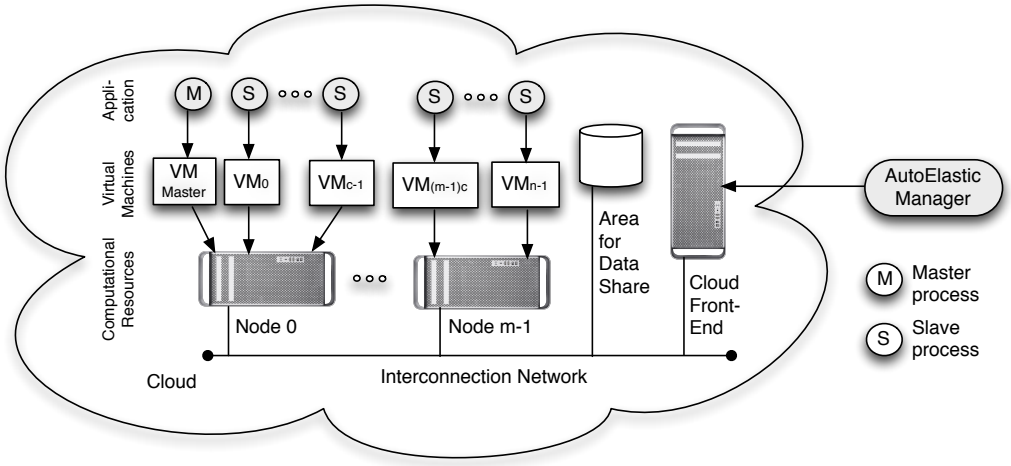


Figure 4. The AutoElastic architecture. While the number of nodes is m, the number of cores in a node is identified by c. The number of VMs running slave processes is n, which can be computed by $c \times m$.

default upper bound of virtual machines is two times the number of VMs used when launching the application. Instead of offering an application-sided elasticity, the use of a manager brings the benefits to resource reorganization in an asynchronous way at the application perspective, not penalizing it on VM (de)allocation actions. However, this non-blocking operation implies in the following question: *How can we notify the application about the resource reconfiguration?* We can achieve this goal through a framework that implements the concept of asynchronous elasticity.

> *Asynchronous elasticity* is a way of asynchronously notifying applications regarding changes on the underlying infrastructure, such as the number of computing instances. For instance, the application is notified as soon as a new computing VM instance (scale out) is available in the system without impairing its normal execution flow.

AutoElastic provides a framework that implements the concept of asynchronous elasticity. One of its key elements to provide asynchronous elasticity in a transparent fashion is a shared data area, which is used to provide interaction between the AutoElastic Manager and the VMs inside the cloud. Shared data areas are a common practice for sharing data between VM instances on cloud infrastructures [17, 18, 19]. They can be implemented by different means such as network file systems, message-oriented middlewares, and tuple spaces. Thus, AutoElastic uses the shared data area as a manner to combine HPC application and cloud elasticity, so providing actions as presented in Table 1.

Table 1. Actions provided through the shared data area.

Action	Direction	Description
Action 1	AutoElastic Manager → Master Process	There is a new resource with c virtual machines which can be accessed using given IP addresses.
Action 2	AutoElastic Manager → Master Process	Request for permission to consolidate a specific node, which encompasses given virtual machines.
Action 3	Master Process → AutoElastic Manager	Answer for Action 2 allowing the consolidation of the specified computing node.

The shared data area provides three types of notifications, as summarized in Table 1. Action 1 is an asynchronous notification sent by the AutoElastic Manager to the application announcing new ready to use computing resources. Figure 5 illustrates the functioning of the AutoElastic Manager when creating a new slave, so launching Action 1 afterwards. Action 2 is required for two reasons: (i) to avoid abruptly finishing a running process, which might lead to data losses; (ii) to ensure that the application will not be aborted due to a sudden interruption of a process. This second rationale is particularly important for MPI applications that execute over TCP/IP networks since they are usually aborted when a process abruptly disconnects. Finally, Action 3 is a decision taken by the master process that avoids inconsistent global state during the application's execution. In other words, once Action 2 has been received, the master process does not dispatch any task to the specific slaves which belong to the node that will be consolidated. The shared data area plays a key role in this process since it keeps all processes updated regarding any resource reconfiguration, allowing a safely adaptation to the new network topology.

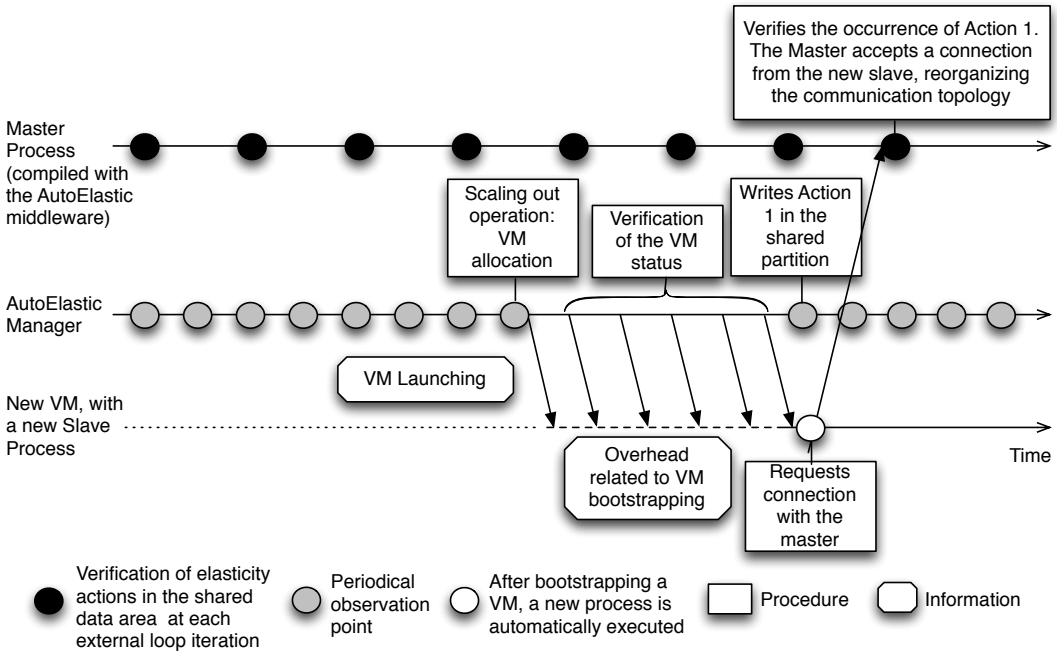


Figure 5. Functioning of the master, the new slave and the AutoElastic Manager to enable the Asynchronous Elasticity.

AutoElastic uses VM replication to provide cloud elasticity for HPC applications [26]. When scaling out the Manager launches new virtual machines using a pre-defined VM template. If the current nodes are working at full capacity, the Manager will first allocate a new computing node to launch the new VMs. The bootstrap of a VM is a time consuming procedure (e.g., boot time of the operating system) that finishes with the execution of a slave process. This slave automatically requests a connection to the master process, completing the asynchronous elasticity cycle. The master process will include the new slaves in the process group without any disruption or interruption on the application's execution. After that, the new slave processes will normally receive tasks from the master. The consolidation (scale in) takes place at a node granularity and not at the VM or process level. This design decision seeks to explore efficiency and energy saving, not using the power of a computing node partially. In fact, it has been

claimed before that the number of VMs or processes inside a node is not the main factor for energy saving, but the fact that the node is turned on or off [27] .

Similarly to previous work [20, 28], AutoElastic performs resource monitoring periodically. Considering a monitoring interval, AutoElastic captures the CPU metric and computes a time series based on the lower and upper thresholds [29]. Particularly, thresholds are largely used in the state-of-the-art of cloud elasticity to drive resource reorganization on CPU-bound applications [1, 2, 4, 28]. AutoElastic uses the concept of moving average over a specific number of load observations to generate a single metric value; so elasticity actions are triggered on situations in which one of the thresholds violates this metric. To accomplish this, we are collecting CPU data using the function LP (Load Prediction), as presented in Equations 1 and 2. $MA(i, j)$ informs the CPU load of a virtual machine j at the observation number i. It performs a moving average considering the last x observations of load C, taking at start point the observation number i. Using this value, we compute an arithmetic average, so establishing an average load for the system by using the function $LP(i)$. In this function, n refers to the number of virtual machines in execution. Action 1 is triggered if LP is larger then the upper threshold, while Action 2 takes place when LP is shorter than the lower threshold. Finally, Equation 3 presents an empirical definition of the cost for execution an application with elasticity. The total number of observations is expressed by z, while $Active\_VMs(i)$ gives us the number of VMs in execution at observation i ($1 \leq i \leq z$). These numbers are important to compare elastic and non-elastic executions of HPC applications. Non-elastic executions have always the same number of VMs for all observations.

$$MA(i,j) = \frac{\sum_{k=i-x+1}^{i} Cjk}{x} \quad where \quad i \geq x \tag{1}$$

$$LP(i) = \frac{\sum_{j=1}^{n} MA(i,j)}{n} \tag{2}$$

$$Cost = app\_time \times \sum_{i=1}^{z} Active\_VMs(i) \tag{3}$$

3.2. Model of Parallel Application

AutoElastic explores data parallelism on iterative message passing applications, which are modeled following the master-slave parallel programming model. This parallel programming model is extensively used in genetic algorithms, the Monte Carlo technique, geometric transformations for 2D and 3D images, asymmetric cryptography and SETI@home-like applications [2]. However, it is worth emphasizing that the framework allows the existing processes of the HPC application to know the identifier of the new instantiated processes, *i.e.*, enabling also a all-to-all communication topology. In other words, it means that AutoElastic supports also applications such as BSP and Divide-and-Conquer.

For developing the communication framework, we investigated the semantics and syntax of both MPI 1.0 and 2.0. While the former statically creates all processes at launching time, the latter supports dynamic process creation and on-the-fly reconfiguration of the connection topology. It means that MPI 2.0 is suitable for elastic environments. The AutoElastic parallel applications follow the MPMD (Multiple Program Multiple Data) principle, where master and slave processes have different executable codes. Each type of binary is mapped to a different VM template. The idea is to offer application decoupling for processes with different purposes, enabling flexibility and making the implementation of elasticity easier. Listing 1 presents a pseudocode of an AutoElastic-supported iterative application. The master code executes a

series of tasks, capturing each one sequentially and parallelizing one-by-one to be processed on slave processes. This behavior can be observed in the external loop (line 2).

Currently, AutoElastic works with the following MPI 2.0-like communications directives: (i) publication of a connection port; (ii) looking for a server, taking as starting point a connection port; (ii) connection request; (iv) connection accept and; (v) disconnection request. Different from the approach in which a master launches processes using the so-called *spawn()* directive, AutoElastic acts in accordance with the second MPI 2.0 approach to support dynamic process creation: Sockets-based point-to-point communication. The launching of a new VM automatically entails the execution of a slave process, which requests a connection to the master automatically, as presented in Listing 2. Here, we emphasize that an AutoElastic-supported application does not need to necessarily rely on the MPI 2.0 API, but only follow the semantics of the communication directives.

Listing 1. Pseudo-language of the the master process

```
1. size = initial_mapping(ports);
2. for (j=0; j< total_tasks; j++){
3.     publish_ports(ports, size);
4.     for (i=0; i< size; i++){
5.         connection_accept(slaves[i], ports[i]);
6.     }
7.     calculate_load(size, work[j], intervals);
8.     for (i=0; i< size; i++){
9.         task = create_task(work[j], intervals[i]);
10.         send_assync(slaves[i], task);
11.    }
12.    for (i=0; i< size; i++){
13.        recv_sync(slaves[i], results[i]);
14.    }
15.    store_results(slave[j], results);
16.    for (i=0; i< size; i++){
17.        disconnect(slaves[i]);
18.    }
19.    unpublish_ports(ports);
20. }
```

Listing 2. Pseudo-language of the of the slave process

```
1. master = lookup(master_address, naming);
2. port = create_port(IP_address, VM_id);
3. while (true){
4.     connection_request(master, port);
5.     recv_sync(master, task);
6.     result = compute(task);
7.     send_assync(master, result);
8.     disconnect(master);
9. }
```

Listing 3. Code to manage elasticity in the master process

```
1. int changes = 0;
2. if (Action == 1){
3.     changes += add_VMs();
4. }
5. else if (Action == 2){
6.     changes -= drop_VMs();
7.     allow_consolidation();// enabling Action3
8. }
9. if (Action ==1 or Action == 2){
10.    reorganize_ports(ports);
11. }
12. size += changes;
```

The `initial_mapping` method (line 1 of Listing 1) is used by the master process to verify the execution configuration, which defines the initial setup of virtual machines, an identifier and

the IP addresses of each process. Taking into account this information, the master knows the number of slaves and creates port names to receive connections from the slave processes. The communication happens asynchronously, where the master sends data to slaves in a non-blocking fashion but receives data from them synchronously. In fact, loop-based programs are convenient to implement cloud elasticity because it is easier to reconfigure the number of resources in the beginning of each iteration without changing the application semantics. Moreover, the job distribution loop is where the global consistent state of the system is kept.

The user must not insert any line about cloud elasticity in the code of the application. AutoElastic middleware manages the transformation of a non-elastic application into an elastic one at the PaaS level by one of the following strategies: (i) polymorphism can overload a method to provide elasticity for object-oriented implementations; (ii) a source-to-source translator can be used to insert code between the lines 1 and 2; (iii) a wrapper for the function in line 3 can be developed for procedural languages. Independent of the strategy, the code required for elasticity is simple, as shown in Listing 3. First, we need to verify if there is a new action from the AutoElastic Manager in the shared data area. If Action 1 has been activated, the master process reads the information regarding the new slaves and knows that it must expect new connections from them. In the case of Action 2, the master removes from its group the processes that belong to a specific node. After doing that, it triggers Action 3.

Although the design of AutoElastic takes into account master-slave applications, the iterative modeling and the use of MPI 2.0-like directives makes it easy to add and remove processes, as well as establishment completely new and arbitrary topologies. At the implementation level, it is possible to optimize connection and disconnection procedures if a particular slave process remains among the active ones in the process list. This improvement can benefit TCP-like connections that require a three way handshake protocol, which might be expensive for some applications.

4. Implementation

We developed an AutoElastic prototype for OpenNebula-based private clouds. The OpenNebula Java API, which was used for developing the AutoElastic Manager, provides the resources required to control both resource monitoring and scaling in and out activities. Moreover, the API is also used to launch parallel applications in the cloud. To run the processes, we created two VM templates, one for the master and another for the slaves. Following, we present some technical decisions in the prototype implementation:

- We used the WS-agreement XML standard[5] to define an SLA, which specify the minimum and maximum number of VMs for the tests;
- The shared data area was implemented through NFS, enabling all VMs inside the cloud infrastructure to access the files. The AutoElastic Manager, which can run outside of the cloud, uses the SSH protocol to access the shared data area on the front-end node;
- The load LP for the monitoring observation number i denoted $LP(i)$ is computed using the moving average of the slave VMs, considering an windows with 3 observations;
- The interval used for monitoring data was 30 seconds;
- Based on the related work (see Section 2), we defined 40% and 80% as the lower and upper thresholds, respectively.

5. Parallel Application and Evaluation Methodology

We developed a numeric integration application to evaluate the gains with and without asynchronous elasticity. The idea was to observe the benefits (e.g., gains in performance, such

[5] https://www.ogf.org/documents/GFD.192.pdf

as reduced execution time) of cloud elasticity for HPC applications. The application computes the numerical integration of a function *f(x)* in a closed interval $[a, b]$. In the implementation, we used the Composite Trapezoidal rule from a Newton-Cotes postulation [30]. The Newton-Cotes formula can be useful if the value of the integrand is given at equally spaced points. Considering the partition of the interval $[a, b]$ into s equally spaced subintervals, each one with length h ($[x_i; x_{i+1}]$, for $i = 0, 1, 2, ..., s-1$). Thus, $x_{i+1} - x_i = h = \frac{b-a}{s}$. The integral of *f(x)* is defined as the sum of the areas of the s trapezoids contained in the interval $[a, b]$, as presented in Equation 4. Equation 5 shows the development of the integral in accordance with the Newton-Cotes postulation.

$$\int_a^b f(x)\,dx \approx A_0 + A_1 + A_2 + A_3 + ... + A_{s-1} \tag{4}$$

where A_i = area of trapezoid i, with $i = 0, 1, 2, 3, ..., s-1$.

$$\int_a^b f(x)\,dx \approx \frac{h}{2}[f(x_0) + f(x_s) + 2.\sum_{i=1}^{s-1} f(x_i)] \tag{5}$$

The values of x_0 and x_s in Equation 5 are equal to a and b, respectively. In this context, s means the number of subintervals. Following this Equation, there are $s + 1$ $f(x)$-like simple equations for obtaining the final result of the numerical integration. The master process must distribute these *s+1* equations among the slaves. Logically, some slaves can receive more work than others when *s+1* is not fully divisible by the number of slaves. Thus, the number of subintervals s define the computational load for each equation.

Aiming at analyzing the parallel application on different input loads, we considered four patterns: Constant, Ascending, Descending and Wave. Table 2 and Figure 6 show the equation of each pattern and the template used in the tests. The iterations in this figure mean the number of functions that are generated, resulting in the same number of numerical integrations. Additionally, the polynomial selected for the tests does not matter in this case because we are focusing on the load variations and not on the result of the numerical integration itself.

Table 2. Functions to express different load patterns. In *load(x)*, x is the iteration index at application runtime.

Load	Load Function	Parameters			
		v	w	t	z
Constant	$load(x) = \frac{w}{2}$	-	1000000	-	-
Ascending	$load(x) = x * t * z$	-	-	0.2	500
Descending	$load(x) = w - (x * t * z)$	-	1000000	0.2	500
Wave	$load(x) = v * z * sen(t * x) + v * z + w$	1	500	0.00125	500000

Figure 7 shows a graphical representation of each pattern. The x axis in the graph expresses the number of functions (one function per iteration) that are being tested, while the y axis informs the respective load. The load means the number of subintervals s between the limits a and b, which in this experiment are 1 and 10, respectively. The larger the number of intervals is, the greater the computational load for generating the numerical integration of the function. For the sake of simplicity, the same function is employed in the tests, but the number of subintervals for the integration varies. Considering the cloud infrastructure, OpenNebula is executed in a cluster with 10 nodes. Each node has two processors, which are exclusively dedicated to the cloud middleware. AutoElastic Manager runs outside the Cloud and uses the OpenNebula API to control and launch VMs. Our SLA was set up for a minimum of 2 nodes (4 VMs) and a maximum of 10 nodes (20 VMs).

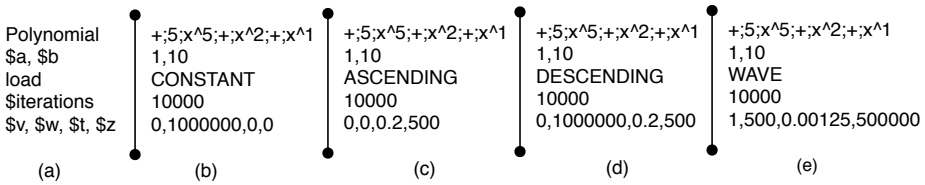


Figure 6. (a) Template of the input file for the tests; (b), (c), (d) and (e) are instances of the template when observing the load functions in Table 2.

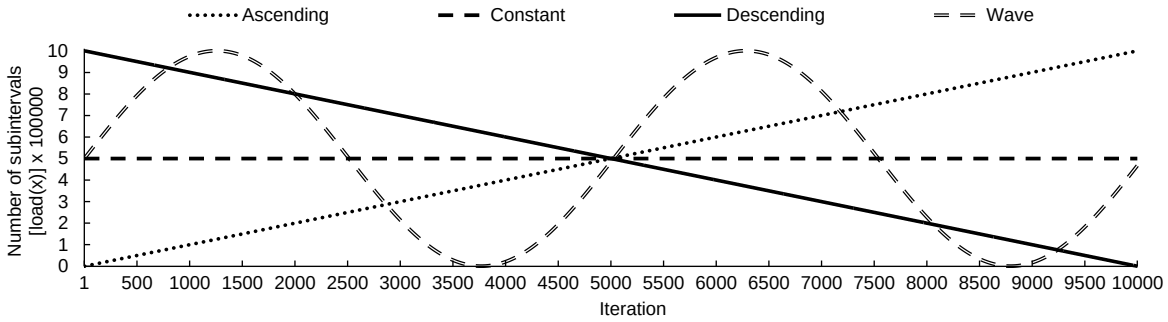


Figure 7. Graphical vision of the load patterns.

6. Evaluation and Discussing Results

We evaluated the numerical application using four load patterns in two scenarios: enabling and disabling cloud elasticity. At each execution, the initial configuration considers 2 nodes, the first executing 2 VMs (2 slave processes) and the second executing 3 VMs (2 slave processes and the master). We collected two metrics, the time (in seconds) to execute the application and the number of load observations performed by AutoElastic during the execution. At each observation i, we have the number of VMs execution on that moment, as well as the result for $LP(i)$. The results can be seen in Table 3. The last column shows the cost according to Equation 3.

Table 3. Results of the executions with and without elasticity support.

Elasticity	Load	Observations with			Total Observations	Time	Cost
		4 VMs	6 VMs	8 VMs			
Disabled	Ascending	84	0	0	84	2426	815136
	Constant	79	0	0	79	2370	748920
	Descending	84	0	0	84	2397	805392
	Wave	84	0	0	84	2444	821184
Enabled	Ascending	31	26	8	65	1978	680432
	Constant	79	0	0	79	2370	748920
	Descending	9	14	33	56	1775	681600
	Wave	9	29	22	60	1895	731470

As can be seen in the Table 3, when elasticity is enabled, the loads Ascending, Descending and Wave used different numbers of VMs during the application execution time. On the other hand, the load Constant used the same configuration in both scenarios with elasticity disabled and enabled. This behavior happened because the $LP(i)$ remained between the lower and upper thresholds, *i.e.*, no elasticity operations were necessary. On the other hand, the execution time and the amount of observations are lower in executions where resource reorganizations happened.

In the Ascending load with elasticity enabled, 47.7%, 40% and 12.3% of the observations

returned 4, 6 and 8 VMs, respectively. The allocation of more VMs along the execution brings a better execution final when compared to the non-elastic execition. This happens because the load grows slowly and takes more time to the $LP(i)$ reach the upper threshold, thus the VM configuration stays with the initial configuration (4 VMs) for a long time. This behavior repeats itself when new resources are available. In the Descending case, 16.1%, 25% and 58.9% of the observations returned 4, 6 and 8 VMs respectively. Here the behavior is opposite to the Ascending load because, in this turn, the resources are allocated in the beginning of the execution and, as the load decreases slowly, it takes more time to reach the lower threshold. Finally, in the Wave load 15%, 48.3% and 36.7% of the observations returned 4, 6 and 8 VMs, respectively. In this case, as the load grows and decrease during the execution, it needs more resources in the beginning and after varies between the scenarios with 6 and 8 VMs.

Figures 8 and 9 illustrate the execution time of the application and the total cost obtained on each scenario. The elastic execution outperforms the non-elastic execution in the Ascending, Descending and Wave patterns, presenting performance gains of 18%, 26% and 22%, respectively. This behavior was also perceived when observing the cost, where AutoElastic with elasticity support resulted in costs approximately 14%, 11% and 10% lower than those with the non-elastic execution for the same mentioned load patterns. Considering that we are allocating more resources on-the-fly to avoid bottlenecks in the application's execution, elasticity helped to reduce the execution times, as can be seen in Table 3. Although using more resources, the gain in the time metric is enough to provide the lower values of cost in favor of the elastic execution. In other words, when compared with the non-elastic execution, AutoElastic uses more resources, which is compensated in terms of execution time.

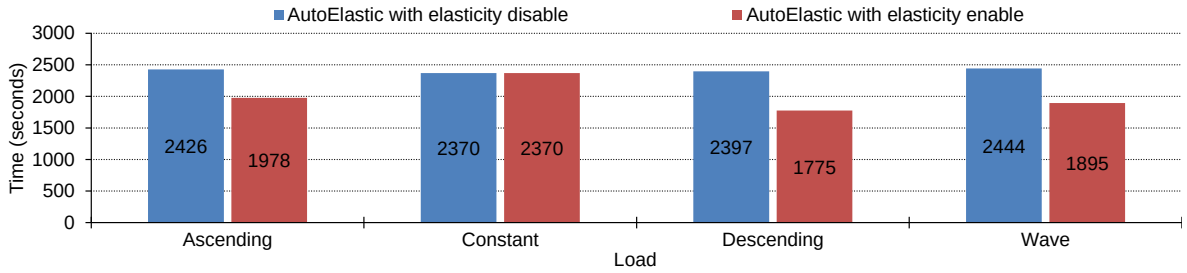


Figure 8. Time to execute the parallel application in the different scenarios and loads.

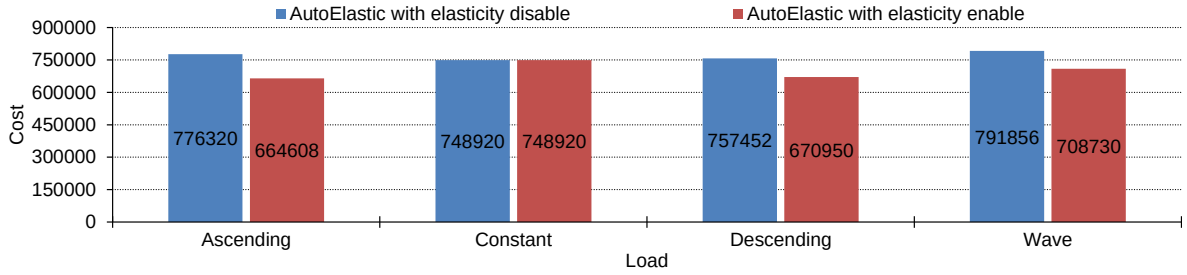
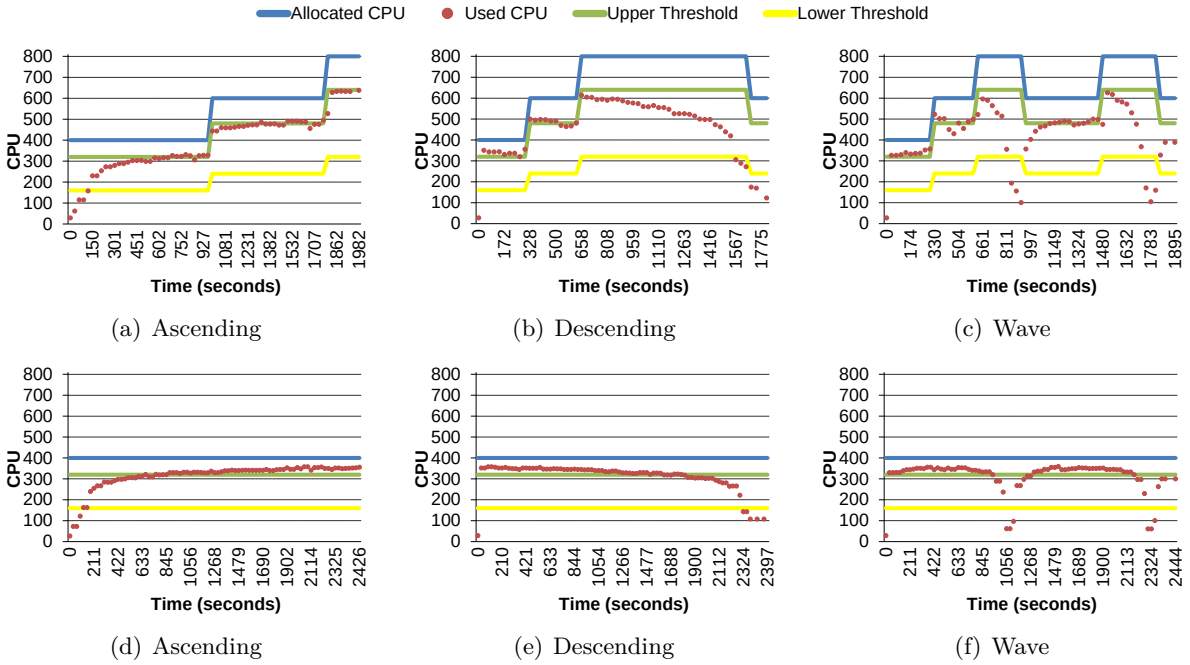


Figure 9. Cost obtained to execute the parallel application in the different scenarios and loads.

Figure 10 depicts a comparison regarding the history of resource allocation when combining load patterns and scenarios. We are not considering the Constant pattern because it does not cause elasticity actions. As expected, we allocate resources in specific moments in the Ascending pattern, while the Descending pattern shows a behavior of allocation in the beginning and a single deallocation in the end of the application. We leave as a future work a deeper analysis of the impact of variable thresholds. More precisely, Figure 10 (b) presents a situation in

which resource management could be improved by increasing the value of the lower threshold. This strategy would imply in a better reactivity and resource usage, since the resources in the descending part will be deallocated sooner. Finally, parts (d), (e) and (f) of Figure 10 present executions in which the CPU demand is close to the theoretical rate available for the application, indicating moments of saturation and so, compromising the application performance.



(a) Ascending (b) Descending (c) Wave

(d) Ascending (e) Descending (f) Wave

Figure 10. CPU behavior: (a), (b) and (c) with the elasticity enabled and (d), (e) and (f) with the elasticity disabled.

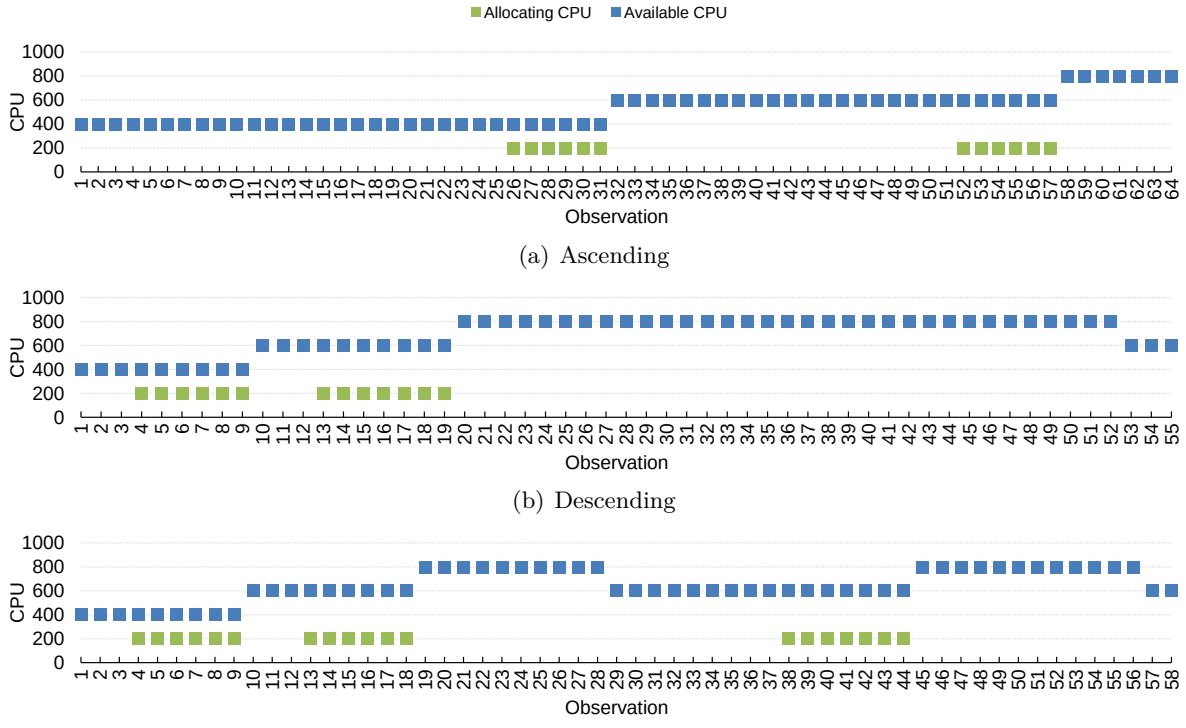
In the environment testbed, the procedure of allocating new resources comprises the transferring of two VMs to a new node in a 100 Mbps network and the initialization of the VMs afterwards. Each VM is based on a template with 700 MBytes in size. During the whole phase of allocating new VMs, the application executes normally with the current resources. The resource reorganization is performed only after completely delivering the new VMs. Table 4 presents the instants in time when new resources were allocated in the tests (see Figure 10). In this table, "VM allocation" represents the instant (including both number of the observation and application time in this instant) in which the $LP(i)$ function violates the threshold so triggering a new resource allocation. The term "VM delivering" represents the moment in which previously allocated resources were deallocated, *i.e.*, detached from the application. The average time between the start of a resource allocation and its delivering to the application is about 214 seconds.

As can be observed in Figure 10, when VMs are being delivered (Table 4), the accumulated CPU load automatically increases as more resources are available. This happens because AutoElastic only notifies the existence of new resources to the application when the the VMs are totally up avoiding pauses in the application execution since the application only needs to connect with the new processes. Figure 11 illustrates the amount of resources that are being delivered, as well as the amount of resources that are being allocated during the application execution. The specific instants in time can be see in Figure 10. Particularly, in blue we are emphasizing the current application resources and in green, we identify the resources that are

Table 4. Analyzing the the time interval between detecting the need to allocate and the delivery of 2 VMs at each elasticity action.

Load	Observation		Time (seconds)		Total Operation Time (seconds)
	VMs Allocation	VMs Delivering	VMs Allocation	VMs Delivering	
Ascending	25	31	752	958	206
Ascending	51	57	1562	1769	207
Descending	3	9	90	295	205
Descending	12	19	388	625	237
Wave	3	9	91	297	206
Wave	12	18	390	597	207
Wave	37	44	1211	1447	236

being allocated (see Table 4 for detail). This figure is important to observe that the resources are only delivered to the application after being completely up; meanwhile, the application executed normally without interruption, maintaining the current number of resources.



(a) Ascending

(b) Descending

(c) Wave

Figure 11. Resource allocations.

Figure 12 depicts the first resource allocation operation among those presented in Figure 11 (a). We can observe three things happening: (i) threshold violation, *i.e.*, the value of $LP(i)$ is greater than the upper threshold; (ii) instantiation of two VMs in a new node; (iii) delivery of the new allocated resources to the application.

7. Conclusion

This article addressed the cloud elasticity for iterative HPC applications through the proposition of the AutoElastic model. AutoElastic self-organizes the number of virtual machines without user intervention, bringing benefits both to the cloud administrator (better energy saving and

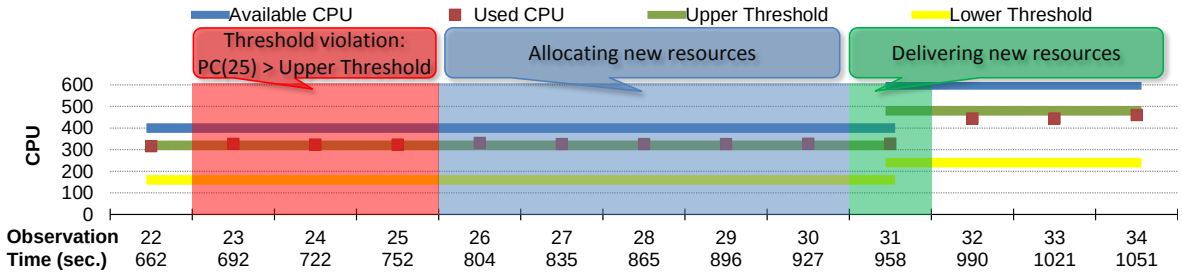


Figure 12. Detailed resource allocation process.

resource sharing among the users) and for the cloud users (who can take profit from a better performance and a quickly application deployment in the cloud). Section 3 presented three problem statements that were addressed as follows:

(i) AutoElastic acts at PaaS level, not requiring that the programmer write elasticity actions and rules in the application code to provide an elastic execution. It also offers asynchronous elasticity, which proved relevant to enable the use of HPC applications in the cloud computing environment.

(ii) The current version of AutoElastic works with master-slave iterative applications, not needing prior information about their behavior. AutoElastic provides a framework totally compatible with tightly-coupled applications, so models such as BSP and Divide-and-Conquer can be adapted in the future to take advantage of cloud elasticity. Concerning the performance gains with cloud elasticity, the evaluation showed that it is possible to reduce about 18% to 26% the execution time of a numerical integration application.

(iii) We are assuming that the user developed an iterative application, providing VM templates both for the master and the slave processes. Moreover, the user has an option to submit an SLA when launching the application. If not provided, AutoElastic takes as default twice the number of VMs at this time for the largest infrastructure.

Our approach for application model is justified by the fact that HPC programs can be developed with the Sockets-like MPI 2.0 programming style. This style allows process connection and disconnection easily, providing an effective use of available resources. AutoElastic offers a reactive and horizontal elasticity, going against the sentence claimed by Spinner et al. [31], who affirm that only vertical scaling is suitable for HPC scenarios due to inherent overhead related to the complementary approach. Thus, we modeled a framework to provide the novel concept of asynchronous elasticity, which turned out as a crucial feature to enable automatic resource reorganization without prohibitive costs. The aforesaid performance results are emphasized when analyzed together with the consumed energy, showing that the AutoElastic's elasticity does not present a forbidding cost.

As a future work, we intend to explore the self-organization of the thresholds in accordance with the application feedback. Finally, as explained earlier, we also plan to extend AutoElastic to contemplate other parallel programming models, including Divide-and-Conquer and BSP.

Acknowledgments
This work was partially supported by the following Brazilian Agencies: CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) e FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul).

References

[1] Lorido-Botran T, Miguel-Alonso J and Lozano J 2014 A review of auto-scaling techniques for elastic applications in cloud environments *Journal of Grid Computing* **12** pp 559–592
[2] Raveendran A, Bicer T and Agrawal G 2011 A framework for elastic execution of existing MPI programs *Proceedings of the 2011 IEEE Int. Symposium on Parallel and Distributed Processing Workshops and PhD Forum* IPDPSW '11 (Washington, DC, USA: IEEE Computer Society) pp 940–947
[3] Han R, Guo L, Ghanem M M and Guo Y 2012 Lightweight resource scaling for cloud applications *Cluster Computing and the Grid, IEEE International Symposium on* **0** 644–651
[4] Ward J S and Barker A 2014 Self managing monitoring for highly elastic large scale cloud deployments *Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing* DIDC '14 (New York, NY, USA: ACM) pp 3–10
[5] Galante G and Bona L C E d 2012 A survey on cloud computing elasticity *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing* UCC '12 (Washington, DC, USA: IEEE Computer Society) pp 263–270
[6] Jennings B and Stadler R 2014 Resource management in clouds: Survey and research challenges *Journal of Network and Systems Management* 1–53
[7] Frincu M E, Genaud S and Gossa J 2013 Comparing provisioning and scheduling strategies for workflows on clouds *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum* IPDPSW '13 (Washington, DC, USA: IEEE Computer Society) pp 2101–2110
[8] Wilkinson B and Allen C 2005 *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers* An Alan R. Apt book (Pearson/Prentice Hall)
[9] Roloff E, Birck F, Diener M, Carissimi A and Navaux P 2012 Evaluating high performance computing on the Windows Azure Platform *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* pp 803–810
[10] Coutinho E, de Carvalho Sousa F, Rego P, Gomes D and de Souza J 2014 Elasticity in cloud computing: A survey *annals of telecommunications - annales des telecommunications* 1–21
[11] Rajan D, Canino A, Izaguirre J A and Thain D 2011 Converting a high performance application to an elastic cloud application *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science* CLOUDCOM '11 (Washington, DC, USA: IEEE Computer Society) pp 383–390
[12] Knauth T and Fetzer C 2011 Scaling non-elastic applications using virtual machines *Cloud Computing (CLOUD), 2011 IEEE International Conference on* pp 468–475
[13] Kumar K, Feng J, Nimmagadda Y and Lu Y H 2011 Resource allocation for real-time tasks using cloud computing *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on* pp 1–7
[14] Michon E, Gossa J and Genaud S 2012 Free elasticity and free CPU power for scientific workloads on IaaS clouds *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on* pp 85–92
[15] Hendrickson B 2009 Computational science: Emerging opportunities and challenges *Journal of Physics: Conference Series* **180** 012013
[16] Tan L, Kothapalli S, Chen L, Hussaini O, Bissiri R and Chen Z 2014 A survey of power and energy efficient techniques for high performance numerical linear algebra operations *Parallel Computing* **40** 559–573
[17] Cai B, Xu F, Ye F and Zhou W 2012 Research and application of migrating legacy systems to the private cloud platform with Cloudstack *Automation and Logistics (ICAL), 2012 IEEE International Conference on* pp 400–404
[18] Milojicic D, Llorente I M and Montero R S 2011 OpenNebula: A cloud management tool *Internet Computing, IEEE* **15** 11–14
[19] Wen X, Gu G, Li Q, Gao Y and Zhang X 2012 Comparison of open-source cloud management platforms: OpenStack and OpenNebula *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on* pp 2457–2461
[20] Chiu D and Agrawal G 2010 Evaluating caching and storage options on the Amazon Web Services Cloud *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on* pp 17–24
[21] Beernaert L, Matos M, Vilaça R and Oliveira R 2012 Automatic elasticity in OpenStack *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management* SDMCMM '12 (New York, NY, USA: ACM) pp 2:1–2:6
[22] Mao M, Li J and Humphrey M 2010 Cloud auto-scaling with deadline and budget constraints *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on* pp 41–48
[23] Martin P, Brown A, Powley W and Vazquez-Poletti J L 2011 Autonomic management of elastic services in the cloud *Proceedings of the 2011 IEEE Symposium on Computers and Communications* ISCC '11 (Washington, DC, USA: IEEE Computer Society) pp 135–140
[24] Zhang X, Shae Z Y, Zheng S and Jamjoom H 2012 Virtual machine migration in an over-committed cloud

Network Operations and Management Symposium (NOMS), 2012 IEEE pp 196–203

[25] Lee Y, Avizienis R, Bishara A, Xia R, Lockhart D, Batten C and Asanovic K 2011 Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators *Computer Architecture (ISCA), 2011 38th Annual International Symposium on* pp 129–140

[26] Kouki Y, Oliveira F A d, Dupont S and Ledoux T 2014 A language support for cloud elasticity management *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* pp 206–215

[27] Baliga J, Ayre R, Hinton K and Tucker R 2011 Green cloud computing: Balancing energy in processing, storage, and transport *Proceedings of the IEEE* **99** 149–167

[28] Imai S, Chestna T and Varela C A 2012 Elastic scalable cloud computing using application-level migration *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing* UCC '12 (Washington, DC, USA: IEEE Computer Society) pp 91–98

[29] Jamshidi P, Ahmad A and Pahl C 2014 Autonomic resource provisioning for cloud-based software *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* SEAMS 2014 (New York, NY, USA: ACM) pp 95–104

[30] Comanescu M 2012 Implementation of time-varying observers used in direct field orientation of motor drives by trapezoidal integration *Power Electronics, Machines and Drives (PEMD 2012), 6th IET International Conference on* pp 1–6

[31] Spinner S, Kounev S, Zhu X, Lu L, Uysal M, Holler A and Griffith R 2014 Runtime vertical scaling of virtualized applications via online model estimation *Proceedings of the 2014 IEEE 8th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*