AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud

Rodrigo da Rosa Righi, *Member, IEEE,* Vinicius Facco Rodrigues, Cristiano André da Costa, *Member, IEEE*, Guilherme Galante, Luis Bona, *Member, IEEE*, Tiago Ferreto

Abstract—Elasticity is undoubtedly one of the most striking characteristics of cloud computing. Especially in the area of high performance computing (HPC), elasticity can be used to execute irregular and CPU-intensive applications. However, the on-the-fly increase/decrease in resources is more widespread in Web systems, which have their own IaaS-level load balancer. Considering the HPC area, current approaches usually focus on batch jobs or assumptions such as previous knowledge of application phases, source code rewriting or the stop-reconfigure-and-go approach for elasticity. In this context, this article presents AutoElastic, a PaaS-level elasticity model for HPC in the cloud. Its differential approach consists of providing elasticity for high performance applications without user intervention or source code modification. The scientific contributions of AutoElastic are twofold: (i) an Aging-based approach to resource allocation and deallocation actions to avoid unnecessary VM (virtual machine) reconfigurations (thrashing) and (ii) asynchronism in creating and terminating VMs in such a way that the application does not need to wait for completing these procedures. The prototype evaluation using OpenNebula middleware showed performance gains of up to 26% in the execution time of an application with the AutoElastic manager. Moreover, we obtained low intrusiveness for AutoElastic when reconfigurations do not occur.

Index Terms—Cloud elasticity, high-performance computing, asynchronism, resource management, self-organizing

1 Introduction

Elasticity is one of the strongest features that distinguishes cloud computing from other approaches of distributed systems [1], [2], [3]. It exploits the fact that resource allocation is a procedure that can be performed dynamically according to the demand for either the service or the user. Elasticity is an essential principle for the cloud model because it not only provides efficient resource sharing among users but also makes it feasible to have a pay-as-you-go computing style. Current state-of-the-art shows that the most common approach for elasticity is the replication of stand-alone virtual machines (VMs), when a particular threshold of a given metric or combination of metrics is reached [3]. The load balancer maintained by the cloud provider manages the service calls (which are usually identified by a URL or IP address) and redirects demands for the most suitable replica [4], [5], [6]. This mechanism was originally developed for dynamic scaling server-based applications, such as web, e-mail and databases, to handle unpredictable workloads, enabling organizations to avoid the downfalls that are involved with non-elastic provisioning (*i.e.*, over- and under-provisioning) [7]. Thus, following this idea, cloud elasticity is also used in different areas, such as video on demand, online stores, BOINC applications, e-governance and Web services [8]. Despite the clear benefits of its adoption, elasticity poses some challenges to the HPC application or service development. Although transparent to the user, the foregoing elasticity mechanism is suitable on loosely coupled programs in which replicas do not establish communication among themselves [9].

Today, multiple cloud computing providers focus on applications that demand high performance computing [10]. Despite the increasing adoption of high-speed network technologies, such as Infiniband and 10 Gigabit Ethernet and hardware-assisted virtualization [11], HPC applications still have difficulty when taking advantage of a variable number of resources for many reasons: (i) the typical development of such applications, that traditionally use MPI (Message Passing Interface) 1.0, involves a fixed number of processes (or threads), so not exploiting the eventual addition of new resources [8]; (ii) despite overcoming the previous limitation by providing dynamic process creation, the applications with MPI 2.0 are not ready, by default, to present an elastic behavior, *i.e.*, programmers must explicitly create or destroy processes during execution, besides managing communicators topology and load balancing by their self [12]; (iii) the fact of either a premature death of a process or a consolidation of a VM that hosts one or more processes from a tightly coupled parallel code can imply the termination of the application execution [8], [18];

- *R. Righi, V. Rodrigues and C. Costa are researchers in the Applied Computing Graduate Program, UNISINOS, São Leopoldo, RS, Brazil E-mail: rrrighi@unisinos.br*
- *G. Galante is with the Computer Science Department, Western Paraná State University, UNIOESTE, Cascavel, Brazil*
- *L. Bona is a researcher in the Department of Informatics, Federal University of Paraná, UFPR, Curitiba, Brazil*
- *T. Ferreto is a member of the School of Computer Science, Pontifical Catholic Univ. of Rio Grande do Sul, PUCRS, Porto Alegre, Brazil*

(iv) the elasticity setup normally requires a previous configuration of lower/upper thresholds, rules and actions, and a definition of a load metric, which can involve both cloud computing expertise and a deep knowledge of the application code [5], [6], [9], [13].

To bypass these limitations in the joint-analysis of cloud elasticity and HPC applications, some approaches impose code rewriting [8], [13], former knowledge of the application phases [14], [15], [16], and the stop-reconfigure-and-go [8] mechanism. Particularly, this last approach is not suitable for short running applications, since the time that the application remains stopped can overcome the possible gains with elasticity. Besides these alternatives, resizing or vertical elasticity [9], [43] can also be employed to to reconfigure CPU, memory or disk parameters of a VM. Although especially pertinent to MPI 1.0 applications, since the number of processes remains unchanged, this technique is bounded by the theoretical characteristics of the node that executes the VM. To the best of our knowledge, as described in Section 2, there is no model that covers distributed resource reconfiguration for HPC systems, addressing concomitantly the aforementioned elasticity problems and transparency in delivering this capability to the user. Considering this background, this article presents an elasticity model called AutoElastic to manage the allocation of virtual machines for HPC applications. AutoElastic acts at the PaaS level of a cloud, not imposing either modifications on the application source code or extra definitions of elasticity rules and actions by the programmer. Furthermore, AutoElastic proposes an operation that does not have any prior knowledge of the application, ignoring, for example, the expected time for concluding each one of its phases. AutoElastic brings two contributions to the state-of-the-art in HPC applications in the cloud:

(i) Use of an **Aging-based technique** [19] for managing cloud elasticity to avoid thrashing on VM allocation and deallocation procedures;
(ii) An infrastructure to provide **asynchronism** on creating and destroying virtual machines, providing parallelism of such procedures and regular application execution.

In addition to AutoElastic, this article also describes both the implementation and evaluation of a prototype that was developed using the OpenNebula [20] middleware. The prototype was used to run a numerical integration application over a private cloud under different situations of load (Ascending, Descending, Wave and Constant). The goal is to show the AutoElastic actions considering the load changes and the impact of both the loads and the asynchronism on the applications performance. The remainder of this article will first introduce related studies in Section 2. Section 3 presents the AutoElastic model, revealing how we developed the aforementioned contributions. Section 4 shows the AutoElastic prototype, the evaluation methodology and a discussion of the results. Finally, Section 5 expresses the final remarks, highlighting the contributions with quantitative data.

2 Related Work

This section is organized to present the perspective of elasticity from both public cloud providers and academic research initiatives, showing the existing gap in the state-of-the-art on managing elasticity for HPC applications. Table 1 shows this compilation. Basically, cloud providers stand to present a robust solution that is ready to use, while academic research shows frameworks, plugins and/or new algorithms for the problem of elasticity in the cloud. In this article, the term elasticity refers to actions for requesting or releasing resources on-the-fly during service runtime [9]. Considering Table 1, the action mode will decide how resources will be dealt with as well as their types [21]. In the horizontal approach, the number of instances (VMs) is increased or decreased. On the other hand, the vertical approach resizes service attributes, such as the CPU, memory or disk capacity. The elasticity management policies are based on [9], which considers two main classifications in accordance with programmer intervention: manual or automatic. The first considers either the use of an application programming interface from the cloud middleware, a graphical or a command-line tool.

In the case of the automatic policy, there are two subdivisions: reactive and proactive. Reactive normally uses rules-condition-action statements and predefined thresholds for elasticity management. On the other hand, a proactive approach employs prediction techniques to anticipate the behavior of the system and to thereby decide the reconfiguration actions. Although the word automatic is used, both policies are expected to require a preliminary configuration from the user viewpoint to set up the initial parameters. According to Galante and Bona [9], there are three methods for providing elasticity: (i) VM replication, (ii) VM migration and (iii) VM redimensioning. The last method concerns the addition of processes for the purpose of exploiting new available resources.

2.1 Commercial and Open Source Initiatives

The commercial and open source initiatives are noted for addressing elasticity either manually, considering the user perception [20], [22], [25], [26], [28], [29], or through previous configurations (with reactive elasticity) [11], [23], [24], [27]. The manual approach is especially common in private clouds. Users can develop their own application for monitoring services that run on VMs, launching elasticity actions when necessary. However, some middleware, such as Amazon AWS, Windows Azure and Nimbus, provide configurable

TABLE 1
An overview of commercial and academic initiatives to address cloud elasticity.

Systems		Level	Action mode	Policy	Objective/Metric	Elasticity Method	Interface/Observations
Commercial, open source initiatives	OpenNebula [20]	IaaS	Horizontal and vertical	Manual	Defined by the user	Replication and migration	CLI, GUI and API
	Eucalyptus [22]	IaaS	Horizontal and vertical	Manual	Defined by the user	Replication and migration	CLI, GUI and API
	Amazon AWS [11]	IaaS, PaaS	Horizontal	Automatic, reactive with preconfiguration	CPU, network, memory and disc	Replication	Amazon AutoScale, CloudWatch and API
	Windows Azure [23]	IaaS, PaaS	Horizontal	Automatic, reactive with preconfiguration	CPU and network	Replication and migration	GUI and API
	Nimbus [24]	IaaS	Horizontal	Automatic, reactive with preconfiguration	CPU	Replication	GUI and Phantom plugin
	OpenStack [25]	IaaS	Horizontal	Manual	Defined by the user	Replication	CLI and API
	CloudStack [26]	IaaS	Horizontal and vertical	Manual	Defined by the user	Replication and migration	CLI, GUI and API
	RightScale [27]	IaaS	Horizontal	Automatic, reactive with preconfiguration	CPU, work queue	Replication	Specialized framework and GUI
Academic research initiatives	PRESS [31]	IaaS	Horizontal	Automatic, reactive	Service Level Objective (SLO)	Redimensioning	Framework for resource management with XEN
	ElasticMPI [8]	PaaS	Vertical	Automatic, reactive	CPU	Replication	Monitoring directives are inserted in the source code
	Work Queue [13]	PaaS	Horizontal	Manual	CPU	Redimensioning	Framework for Windows Azure and Amazon AWS
	COS [5]	PaaS	Horizontal	Automatic, reactive	CPU and network	Replication and Migration	Management on operating system-level
	Ming, Li e Humphrey [6]	IaaS	Horizontal	Automatic, proactive	Conclusion time of batch jobs	Replication	System that works over Windows Azure
	Lightweight Scaling [7]	IaaS	Horizontal and vertical	Automatic, reactive	CPU and memory	Replication, redimensioning	Framework for resource management
	Kingsfiher [21]	IaaS	Horizontal	Automatic, reactive and proactive	Budget (cost)	Replication and migration	Executes over OpenNebula
	Moreno e Xu [33]	IaaS	Horizontal and vertical	Automatic, proactive	Energy	Replication and migration	Framework for resource management
	Scattered [34]	IaaS	Horizontal	Automatic, proactive	CPU and network	Migration	Framework for resource management
	Sandpiper [35]	IaaS	Horizontal and vertical	Automatic, reactive an proactive	CPU, network and memory	Redimensioning and migration	Framework that works over XEN
	Elastack [36]	IaaS	Horizontal	Automatic, reactive	CPU	Replication	Plugin that works over OpenStack
	Elastic Queue Service [37]	IaaS	Horizontal and vertical	Automatic, reactive	Number of requests	Replication	Framework that works over Amazon AWS
	Elastic Site [24]	IaaS	Vertical	Automatic, reactive with preconfiguration	Number of requests	Replication, redimensioning	Resource manager for Nimbus
	Suleiman [38]	IaaS	Vertical	Automatic, reactive	CPU	Replication	Use Amazon AWS and CloudWatch
	Kaleidoscope [40]	IaaS	Horizontal	Automatic, reactive	CPU and memory	Replication (cloning)	Micro-elasticity by VM cloning

subsystems for service monitoring and elasticity management. Users must complete the rules and the limits of a metric to be monitored as well as the conditions and actions for reconfiguration. AWS provides an API and a graphical tool for these tasks. The monitoring is performed by CloudWatch, and the elasticity itself is managed reactively by Autoscaling [11].

Windows Azure offers a library, called Autoscaling Application Block, in which the user must add the application project for enabling the elasticity feature. This approach allows the programmer to use performance counters and to write elasticity rules. AzureWatch is another way of controlling the number of instances; it is a framework that works with CPU usage, history, mean response time and other data regarding the VM [23]. Another possibility in AzureWatch is to employ explicit action rules. The Nimbus system, in turn, works as a batch job scheduler for Amazon EC2-based cloud environments. Nimbus uses Phantom to monitor the behavior of resources based on an observed metric. Phantom allows setting of the minimum and maximum number of VMs for a service execution, the number of VMs that will be launched when an upper limit is found and the number of VMs that will be consolidated when attaining a lower limit.

2.2 Academic Research Initiatives

The academic research initiatives aim to fill the gaps or improve the already existing elasticity approaches. In this way, the following subgroups can be extracted: (i) parallel and high performance computing [1], [5], [6], [7], [8], [13], [14], [16], [32], [36], [41]; (ii) budget-oriented (financial costs) [6], [21]; (iii) federated-based cloud computing [5], [24], [30]; (iv) predictive models for detecting behavior patterns on cloud services [6], [21], [31], [33], [34], [35]; (v) energy savings and overbooking of VMs over physical machines [14], [33]; (vi) real-time computing, addressing deadlines for the

services [6], [8], [33], [37]; (vii) message-oriented middleware solutions [37]; and (vii) web-based workload and applications for both business and transactional areas [4], [32], [35], [38], [39].

In the following, the HPC subgroup is expanded due to its tight relation with this work. ElasticMPI offers elasticity for MPI applications by stop-and-relaunching the application with a newer resource configuration [8]. The system assumes that the user knows in advance the expected conclusion time for each phase of the program. The monitoring system can detect that the current configuration cannot fulfill the given deadline and adds more resources. Furthermore, the approach of ElasticMPI imposes changes in the application source code by inserting monitoring directives. Imai et al. proposes an operating system-level approach called COS [5], which provides a generic framework based on an actors-oriented programming language SALSA. When observing that all VMs are overloaded, migrations can occur both at the VM (from one node to another) and actor (from one VM to another) levels. Nevertheless, COS works only with applications that are strictly composed by SALSA migration-capable components.

Ming, Li and Humphrey [6] present the notion of self-scalability, in which the number of VM instances fluctuates in accordance with the service workload. Once the program has deadlines for executing each one of its phases, the proposal works with resources and VM allocations to meet those limits. Martin et al. [32] present a typical scenario of requests on a cloud service that acts as a load balancer. The elasticity changes the amount of worker VMs according to the service demands. Elastack, in its turn, is a system that runs over OpenStack and uses Serpentine to detect changes in the environment [36]. When finishing the workload on each instance, Serpentine warns the balancer that the target instance can be consolidated.

Kumar et al. [41] address elasticity on batch jobs. The application execution is not changed, but it works with malleable launching times. The anticipation of work is performed based on monitoring data. Michon et al. [16] work with different job mapping strategies for VMs: (i) all jobs to one VM; (ii) one job per VM and (iii) first fit. The input is marked by a series of batch jobs that must be mapped to VMs. The authors perform tests with fixed data regarding the set of jobs, execution times and time in queues. Moreover, they performed tests only with Bag-of-Tasks applications. Weiwei et al. [1] uses the CloudSim simulator to enable their threshold-based dynamic resource allocation scheme. Instead of performing the monitoring activity at fixed intervals, their proposal turns the current interval malleable in accordance with the applications regularity. Despite this advantage, the code execution must wait when blocking primitives for scaling up and down. Furthermore, there is no peak treatment when reaching the thresholds.

According to the academic initiatives presented in Table 1, a technique that is widely used for load balancing is VM live migration, which is common in traditional hypervisors such as Xen and KVM. The table also reveals that the primary metric of work is the CPU, including its clock and load, process execution times, number of instructions per job or process, redimensioning to the percentage of CPU usage and makespan. In addition, the elasticity is further explored at the IaaS level and in a reactive fashion. The initiatives are not uniform regarding the percentage to be used on the thresholds: 70% [4], 75%[5], 80% [40] and 90% [36], [38]. These values address upper limits that, when exceeded, trigger horizontal or vertical elasticity.

2.3 Analysis and Research Opportunities

This subsection presents some weak points regarding the studied systems in Sections 2.1 and 2.2. Systems that are available in the Web are characterized as general purpose initiatives and typically require user intervention on a previous elasticity configuration. In summary, we highlight the following weak points: (i) no analysis of sporadic peak situations when reaching a threshold [1], [32], [36]; (ii) need to change the application source code [8], [13], [42]; (iii) use of proprietary components that are not available as programming libraries for well-known operating systems such as GNU-Linux, Windows and MacOS [5], [8], [36]; (iv) need to know application behavior beforehand, such as the expected execution time of the components [8], [14], [15], [16]; (v) resource reconfiguration with a stop-reconfigure-and-go approach [8]; and (vi) communication among VMs at a constant rate [34].

Considering the specific area of parallel applications and elasticity, we highlight the following initiatives: [8], [13], [17], [32], [42], [43]. Among them, three initiatives execute iterative applications, where each new phase means that there is a new effort for redistributing the tasks to slaves [8], [13], [17]. The elasticity in [13] is offered manually, where the user captures monitoring data by using the framework proposed by the authors. Despite the gaps mentioned before regarding ElasticMPI [8], the strength is to offer (with some limitations) elasticity over existing MPI programs (versions 1 and 2). Jackson et al. [17] execute master-slave programs from NERSC as a benchmark for measuring the performance of cluster configurations on Amazon EC2. The solution proposed by Martin et al. [32] concerns the efficiency of addressing requests on a web server. This solution delegates and consolidates instances according to both the time between the request arrivals and the load on the worker VMs. Spinner et al. [43] propose vertical scaling middleware for individual VMs that run HPC applications. They argue that the horizontal approach is prohibitive in the HPC scope because, following

them, a VM instantiation takes at least 1 min to be ready. Finally, elasticity is offered at the API-level, where the user manages resource reconfigurations by himself/herself [42]. As said earlier, this strategy requires cloud expertise of the programmer and would not be portable among different cloud deployments.

As being a time-consuming operation, the moment of an scaling out operation must be carefully analyzed to properly execute HPC applications in the cloud. Both Amazon AWS [11] and Windows Azure [23] expect x consecutive load observations outside the margin of a threshold to launch elasticity actions. Besides x and lower/upper thresholds, they also offer a parameter named cool-down, that refers to a period after an scaling operation which new elasticity actions are prohibited. In general, academic initiatives present the following strategies: (i) developed to run over Amazon AWS, they inherit the aforesaid conditions [11], [13], [38], [39]; (ii) as Amazon and Azure, the initiatives [5] and [15] also use a parameter x to represent the number of times that a thresholds must be exceeded; (iii) wait for y seconds after surpassing a threshold to enable a new elasticity action [3], besides the use of a cool-down period; (iv) after taking a single load observation, elasticity takes place if a threshold is exceeded [2], [16], [36], [40]; (v) an elasticity action is automatically triggered when occurring any violation to an SLA (Service Level Agreement) or SLO (Service Level Objective) [4], [21], [32], [37]; (vi) use of application knowledge, such as deadlines, to conclude parts of a workload, so if the deadline was reached, a new resource is added to improve the system performance [6], [8]. Strategies iv and v can present the thrashing problem (*i.e.*, oscillations) on VM allocation and deallocation actions, since the application can be crossing a sudden peak. In addition, strategies i, ii and iii may not be reactive, presenting a false-negative scenario, since a either x or y can neglect the real need of an elasticity action. Finally, adopting one of these three strategies, the user will need a deep knowledge about the application code to setup the elasticity parameters, which may not be a trivial task.

3 AutoElastic: PaaS-Based Model for Automatic Elasticity Management

This section describes AutoElastic, which is an elasticity model that focuses on high performance applications. AutoElastic is based on reactive elasticity, in which the rules are defined without user intervention. The definition of elasticity rules is not a trivial task for several reasons. It involves the setup of one or more thresholds, types of resources, number of occurrences, definitions of peaks and monitoring windows. Furthermore, it is not uncommon to have situations in which optimized elasticity rules control the behavior of one application and present poor performance over other applications. Additionally, the scale-out elasticity triggers resource allocations that present a direct impact in the cost of cloud usage. Thus, the cost/benefit ratio should be carefully evaluated. AutoElastic's approach provides elasticity by hiding all of the resource reconfiguration actions from programmers, executing without any modifications in the application's code. In particular, AutoElastic addresses applications that do not use specific deadlines for concluding the subparts. Its differential approach covers two topics: (i) efficient control of VM launching and consolidation totally transparent to the user and (ii) a mechanism to execute HPC programs on the cloud in a non-prohibited way. Neither topic was found when analyzing related work.

3.1 Design Decisions

Figure 1 depicts the main idea of AutoElastic. Acting at the PaaS level, it presents a middleware that can be seen as a communication library used for compiling the application. Moreover, it comprises an elasticity manager that controls resource reconfiguration on behalf of the cloud and user. AutoElastic was modeled with the following requirements and design decisions in mind: (i) users do not need to configure the elasticity mechanism; however, programmers can inform an SLA (Service Level Agreement) with the minimum and maximum number of VMs to run the application; (ii) programmers do not need to rewrite their application to profit from the cloud elasticity; (iii) the investigated scenario concerns a non-shared environment and the execution of a single application; (iv) AutoElastic offers a reactive, automatic and horizontal elasticity, following the replication strategy to enable it; (v) because it is a PaaS-based framework, AutoElastic comprises tools for transforming a parallel application to an elastic application transparently to users; and (vi) AutoElastic analyses the load peaks and sudden drops for not launching unnecessary actions, avoiding a phenomenon known as thrashing.

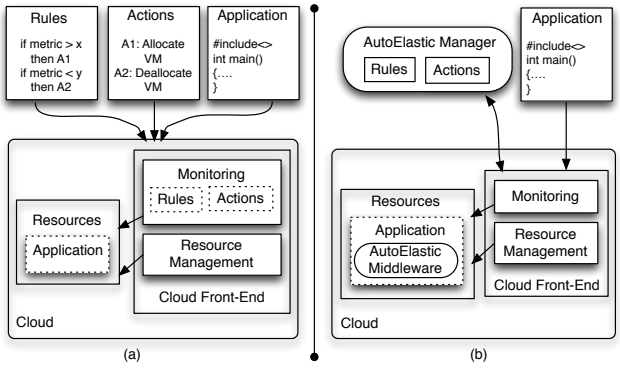

Fig. 1. (a) Approach adopted by Azure and Amazon AWS, in which the user must define elasticity rules and configure metrics beforehand; (b) AutoElastic idea.

AutoElastic Manager considers data about the virtual machines load as input for rules and actions,

thus acting on reconfiguring iterative-based master-slave parallel applications. Although trivial, this type of construction is used in several areas, such as genetic algorithms, Monte Carlo techniques, geometric transformations in computer graphics, cryptography algorithms and applications that follow the Embarrassingly Parallel computing model [8]. At a glance, the idea of offering elasticity to the user in an effortless manner while considering only the performance perspective, which is mandatory on HPC scenarios, is the main justification for the decisions that we adopted.

3.2 Architecture

Figure 2 illustrates AutoElastic's components and how VMs are mapped on homogeneous nodes. AutoElastic Manager can either be assigned to a VM inside the cloud or act as a stand-alone program outside the cloud. This flexibility is achieved because of the use of the API offered by the cloud middleware. Here, we do not present the AutoElastic middleware because it is integrated with each application process. Regarding the application, there is a master and a collection of slave processes. Considering that parallel applications are commonly CPU-intensive, AutoElastic uses a single process on each VM and c VMs per node for processing purposes, where c means the number of cores in a specific node. This approach is based on the work of Lee et al. [44], where they seek to explore better efficiency for HPC applications.

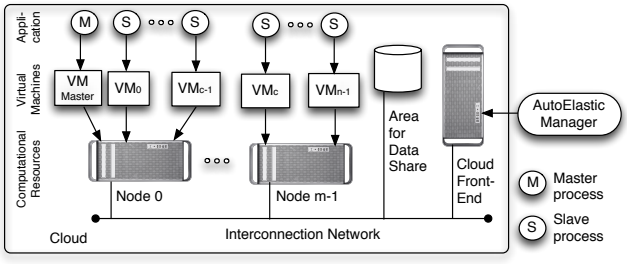

Fig. 2. AutoElastic architecture. Here, c denotes the number of cores inside a node, m is the number of nodes and n refers to the number of VMs running slave processes, being obtained by $c.m$.

AutoElastic Manager monitors the running VMs periodically and verifies the necessity for a reconfiguration based on the load thresholds. The user can pass to the manager a file that has an SLA that specifies both the minimum and the maximum number of VMs to execute the application. AutoElastic follows the XML-based WS-Agreement standard to define an SLA document. If this file is not provided, it assumes that the maximum number of VMs is twice the amount informed when launching the parallel application. Concerning the fact that the manager, not the application itself, increases or decreases the number of resources brings the following benefit: the application is not penalized by the overhead of synchronous allocation or deallocation of resources. Nevertheless, this asynchronism leads to a question: *How do we notify the application about the resources reconfiguration?*

To answer this question, AutoElastic provides a shared data area to enable communication between VMs and the AutoElastic Manager. This area is private to the VMs or nodes inside the cloud. It can be implemented, for example, through NFS, message-oriented middleware (such as JMS or AMQP) or tuple spaces (such as JavaSpaces). The use of a shared area for interaction among virtual machines is common practice when addressing private clouds [20], [25], [26]. When modeling the AutoElastic Manager outside the cloud, we can use the cloud middleware supported API to obtain monitoring data about the VMs. To write and read elasticity actions to and from the shared area, AutoElastic uses a remote secure copy utility that targets a shared file system partition in the cloud front-end. For example, the SSH package commonly offers the SCP utility for this functionality.

3.3 Elasticity Model

The elasticity is guided by actions that are enabled through the use of the shared area. Table 2 shows the actions that are defined to enable the communication between the AutoElastic Manager and Master Process. After obtaining a notification for Action1, the master can establish connections with the slaves in the newer virtual machines. Action2 is appropriate for the following reasons: (i) not finalizing the execution of a process during its processing phase and (ii) to ensure that the application as a whole is not aborted with a sudden interruption of one or more processes. In particular, the second reason is important for MPI 2.0 applications running over TCP/IP, which is interrupted when detecting the premature death of any process. Action3 is generated by the master process, which ensures that other processes are not in processing phases; then, specific slaves can be disconnected.

In practice, the communication between AutoElastic and the master does not require a shared data area. Precisely, the shared area only makes sense for the master in the current master/slave approach because only the master will address process reorganization. However, this approach was adopted to make AutoElastic more flexible. The use of a shared area that the master, slaves and AutoElastic Manager have access to is provided in the future support of Bulk Synchronous Parallel and Divide-and-Conquer programs. In these programming models, all of the processes must be informed about the current resources.

AutoElastic uses the replication strategy to enable elasticity. Considering the scaling out operation, the AutoElastic Manager launches new virtual machines based on a template image for the slave processes. They are instantiated into a new node, which char-

TABLE 2
Actions provided by AutoElastic for communication between AutoElastic Manager and Master Process.

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 compute node.

acterizes a horizontal elasticity approach. The bootstrap procedure on each allocated VM ends with the automatic execution of the slave process, which attempts to establish communication with the master. In contrast to the standard use of the replication technique mentioned in Section 1, our framework provides a shared area to receive and store application and network parameters, making possible a totally arbitrary communication style among the processes.

Scale-out operations (increase/decrease the number of VMs) are asynchronous from the application perspective because the master and slave processes can continue their execution normally. In fact, the instantiation of VMs is performed by the AutoElastic Manager and, only after they achieve a running state, the manager notifies the master using the shared data region. Figure 3 illustrates this situation, making clear the concomitance between the master process and the procedure of VM allocation. Spinner et al. [43] emphasize that horizontal elasticity is prohibitive for HPC environments, but here we are proposing asynchronous elasticity as an alternative, to join both themes in an efficient way. Regarding the consolidation policy, the work grain is always a compute node and not a specific VM; as a result, all VMs that belong to a node will go down with such an action. This approach is explained by the fact that a node is not shared among other users and that this approach saves on energy consumption. In particular, Baliga et al. [45] claim that the number of VMs in a node is not a major influential factor on the energy consumption; instead, the fact that the node is on or off is influential.

As in [5], [11], AutoElastic monitoring activity is performed periodically. The manager captures the values of the CPU load metric on each VM and uses them in comparison with both minimum and maximum thresholds. Each monitoring observation entails the capture of this metric and its recording (allowing the use of temporal information) in a log-rotate fashion. Data are saved in the AutoElastic Manager repository. Figure 4 presents the reactive-based model for the AutoElastic elasticity. Although the system contemplates three actions, this figure presents only the actions that are triggered by the AutoElastic Manager. In the conditions part, a function called *System_Load(j)* is an arithmetic average of the *LP(i,j)* (Load Prediction) of each node, where i stands for a VM, j is the current observation and n is the number of VMs that are running a slave process. Equations 1 and 2 present the aforesaid functions. *LP(i,j)* operates with a parameter-denoted *window* that defines the number of values to be evaluated in the time series. Here, t means the index of the most recent observation, *i.e.*, the value used in calling *System_Load(j)* (j=t).

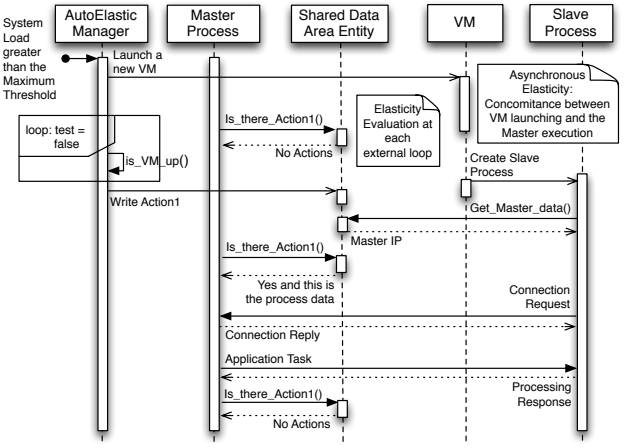

Fig. 3. Sequence diagram when detecting a system load greater than the maximum threshold.

RULE1: if CONDITION1 then ACTION1
RULE2: if CONDITION2 then ACTION2

CONDITION1: System_load(j) $>$ threshold1, where j means the last monitoring observation
CONDITION2: System_load(j) $<$ threshold2, where j means the last monitoring observation

ACTION1: Allocates a new node and launches c VMs on it, where c is the number of cores inside the node
ACTION2: Finalizes the VM instances that are running inside a node and performs the node consolidation afterwards

Fig. 4. Reactive-driven elastic model of AutoElastic.

$$System\_Load(j) = \frac{1}{n}. \sum_{i=0}^{n-1} LP(i,j) \quad (1)$$

such that

$$LP(i,j) = \begin{cases} \frac{1}{2}L(i,j) & if j = t - window + 1 \\ \frac{1}{2}LP(i,j-1) + \frac{1}{2}L(i,j) & if j \neq t - window + 1 \end{cases} \quad (2)$$

where t is the value of j when *System_load(j)* is called, and *L(i,j)* refers to the CPU load of VM i at the monitoring observation that is numbered j.

LP uses a *window* to operate the Aging concept [19]. It employs an exponentially weighted moving average in which the last measure has the strongest influence on the load index. In this context, if we decide that the weight of each observation must not be shorter than 1%, the *window* can be set to 6 because $\frac{1}{2^6}$ and $\frac{1}{2^7}$ express the 1.56% and 0.78% percentages, respectively. We employ the idea of Aging when addressing peak situations, especially to avoid either false-negative or

false-positive elasticity actions. For example, assuming a maximum threshold of 80%, a *window* equal to 6 and monitoring values such as 82, 78, 81, 80, 77 and 93 (the most recent one), we have $LP(i,j) = \frac{1}{2}.93 + \frac{1}{4}.77 + \frac{1}{8}.80 + \frac{1}{16}.81 + \frac{1}{32}.78 + \frac{1}{64}.82 = 84.53$. This value enables resource reconfiguration. In contrast to the AutoElastic approach, Imai et al. [5] expect x consecutive observations outside the margin of the threshold. In this case, the use of x equal to either 2, 3 or 4 samples does not imply elasticity, and the system will operate at its saturated capacity. In addition to this false-negative scenario, a false-positive happens when an application presents an unexpected peak or drop, so triggering an unnecessary scaling out or scaling in operation. Thus, AutoElastic's idea for thrashing avoidance contemplates the use of time series and weighted moving average to smooth possible noises on load observations.

3.4 Parallel Application Model

AutoElastic exploits data parallelism to handle iterative message-passing applications that are modeled as a master-slave. In this way, the composition of the communication framework began by analyzing the traditional interfaces of MPI 1.0 and MPI 2.0. In the former, process creation is given in a static approach, where a program starts and ends with the same number of processes. MPI 2.0 provides features that enable elasticity because it offers both the dynamic creation of new processes and the on-the-fly connection with other processes in the topology. AutoElastic parallel applications are designed according to the MPMD model (Multiple Program Multiple Data) [18], where multiple autonomous VMs execute simultaneously one type of program: master or slave. This decoupling helps to provide cloud elasticity because a specific VM template is generated for each program type to enable a more flexible scaling-out operation.

Figure 5 (a) presents a master-slave application that is supported by AutoElastic. As stated, it has an iterative behavior in which the master has a series of tasks, sequentially distributing them through the slave processes. The distribution of tasks is emphasized in the external loop of Figure 5 (a) (lines 2 to 20). Based on the MPI 2.0 interface, AutoElastic works with the following groups of programming directives: (i) publication of connection ports, (ii) searching for a server using a specific port, (iii) connection accept, (iv) requesting a connection and (v) disconnection. Unlike the approach in which the master process dynamically launches other processes (using the spawn call), the proposed model operates according to the other MPI 2.0 approach for dynamic process management: point-to-point communication with Socket-like connections and disconnections [12]. The launching of a new VM automatically entails the execution of a slave process, which requests a connection to the master automatically. We emphasize that a program with AutoElastic does not need to follow the MPI 2.0 API and instead follows the semantics of each aforesaid directive.

```
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.          conection_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. }
```

(a)

```
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.  }
```

(b)

```
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;
```

(c)

Fig. 5. Application model in pseudo-language: (a) Master process; (b) Slave process; (c) elasticity code to be inserted in the Master process at PaaS level by using either method overriding, source-to-source translation or wrapper technique.

Communications between master and slaves follow the asynchronous model, where sending operations are nonblocking and receiving operations are blocking (see lines 5 and 7 of Figure 5 (b)). The method in line 1 of the master process checks either a configuration file or arguments passed to the program to obtain the virtual machine identifiers and the IP address of each process. Based on the results, the master knows the number of slaves and creates port numbers to receive connections from each slave. The publishing of the ports occurs in the method of line 3. Programs with an outer loop are convenient for elasticity establishment because in the beginning of an iteration, it is possible to make resource reconfigurations without changing both the application syntax and semantics [43]. The transformation of the application shown in Figure 5 in an elastic situation is performed at the PaaS level by applying one of the following methods: (i) in an object-oriented implementation, overriding the *publish_ports()* method for elasticity management; (ii) use of a source-to-source translator that inserts the elasticity code between lines 3 and 4 in the master code; (iii) development of a wrapper for procedural languages to change the method in line 3 of the master code transparently.

The additional code for enabling elasticity checks in the shared directory to determine whether there is any new data from the AutoElastic Manager (see Figure 5 (c)). In the case of Action1, the manager already set up new VMs, and the application can add data of the new slaves in the slaves set. In case Action2 takes place, the application understands the order from the manager and reduces the number of VMs (and consequently, the number of processes in the parallel application) and triggers Action3. Although the initial focus of AutoElastic is in iterative master-slave applications,

the use of MPI 2.0-like directives makes the inclusion of new processes and the reassembly of arbitrary communication topologies easier. At the implementation level, it is possible to optimize the connections if a process remains in the list of active processes. This circumstance is pertinent over TCP networks, which use a three-way handshake protocol known as an overhead source when connecting two end points.

4 EVALUATION

This section presents details about the AutoElastic prototype and a description of the evaluated application. Moreover, it covers the results and their linking with the contributions aforementioned in Section 1.

4.1 Prototype Implementation

We implemented an AutoElastic prototype for private clouds using OpenNebula v,4.2. We implemented the AutoElastic Manager and the parallel application in Java. Two image templates for the virtual machines were provided: one for the master process and another for the slaves. The manager uses the Java-based OpenNebula API for both monitoring and elasticity activities. In addition, this API is also used for launching a parallel application in the cloud, which is associated with an SLA that can be supplied by the user. The SLA follows the XML-based WS-Agreement standard and reports the minimum and maximum number of VMs for testing the application.

There are some technical decisions that are imposed in the implementation of the AutoElastic prototype: (i) utilization of NFS (Network File System) to implement the AutoElastic module responsible for providing a private area for data sharing inside the cloud; (ii) AutoElastic employs periodic monitoring, which is associated with a period of 30 s (OpenNebula lower bound index); and (iii) Contemplating related work, 80% and 40% were used for maximum and minimum thresholds. Looking at decision (i), the manager uses the binary SCP from the SSH package to obtain or place files from/to the cloud front-end. These files are in a shared folder that is enabled with NFS and represents elasticity actions and process data. We follow this implementation because the AutoElastic Manager cannot access the NFS directly, unless it is placed inside the cloud.

4.2 Parallel Program and Environment Setup

The application used in the tests computes the numerical integration of a function *f(x)* in a closed interval $[a, b]$. It was implemented using the Composite Trapezoidal rule from a Newton-Cotes postulation [46]. 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 3. Equation 4 shows the development of the integral in accordance with the Newton-Cotes postulation. This equation is used to develop parallel application modeling.

$$\int_a^b f(x)\,dx \approx A_0 + A_1 + A_2 + A_3 + ... + A_{s-1} \quad (3)$$

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

$$\int_a^b f(x)\,dx \approx \frac{h}{2}[f(x_0) + f(x_s) + 2.\sum_{i=1}^{s-1} f(x_i)] \quad (4)$$

The values of x_0 and x_s in Equation 4 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 will define the computational load for each equation.

Aiming at analyzing the parallel application on different input loads, four patterns were developed and named: Constant, Ascending, Descending and Wave. Table 3 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 placing attention on the load variations and not on the result of the numerical integration itself.

TABLE 3
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

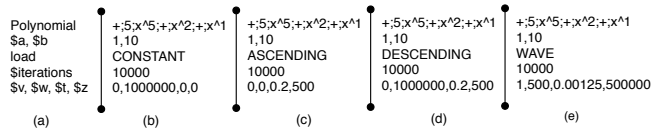

Fig. 6. (a) Template of the input file for the tests; (b), (c), (d) and (e) are instances of the template when observing the load functions in Table 3.

Figure 7 shows a graphical representation of each pattern. The x axis in the graph of Figure 7 expresses

the number of functions (one function per iteration) that are being tested, while the y axis informs the respective load. Again, 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). Finally, we adopt a *window* parameter that is equal to 6 in the load prediction (*LP*) function to ensure that the weight of each observation is larger than 1% (see Subsection 3.3 for detail).

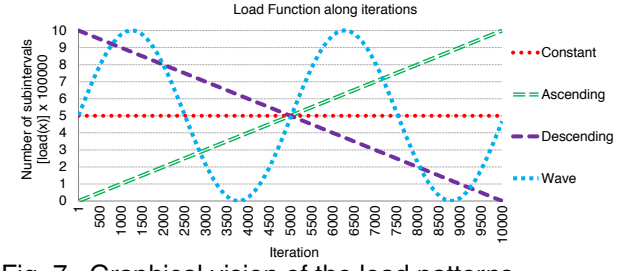

Fig. 7. Graphical vision of the load patterns.

4.3 Results and Discussion

We divided this subsection in two parts to debate elasticity behavior. Following, we present the goals and the metrics of each part:

- Performance and Elasticity Viability: the objective consists in presenting performance results when combining starting configurations, load patterns and two scenarios: (i) use of AutoElastic, enabling its self-organizing feature for dealing with elasticity; (ii) use of AutoElastic, managing load situations but without taking any elasticity action. The metrics measured in this evaluation are: (i) the time in seconds to execute the application; (ii) the number of load observations of the AutoElastic manager; (iii) a comparison index defined as *Resource* and; (iv) a *Cost* function. This last, presented in Equation 5, means an adaptation of the cost of a parallel computation [18] for elastic environments. Instead of using the number of processors, here *Resource* is evaluated empirically taking into account both the number of observations as *Obs(i)* and the quantity of VMs i employed in each observation (see Equation 6). The AutoElastic's goal is to offer an elasticity model that is costly viable. In other words, a configuration solution may be classified as a bad if it is capable to reduce the execution time by the half with elasticity, but spends four times more energy, increasing the costs for that. Therefore, considering the values of the cost in Table 4, the objective is to preserve the truth of Inequality 7.

$$Cost = App\_Time \times Resource \quad (5)$$

such that

$$Resource = \sum_{i=1}^{n} (i \times Obs(i)) \quad (6)$$

then, the goal is to obtain

$$Cost_{\alpha} \leq Cost_{\beta} \quad (7)$$

where α and β refer to the AutoElastic execution with and without elasticity support, respectively.

- Analyzing Asynchronous Elasticity: the goal is to present the benefits of asynchronous elasticity and the master process behavior when resource reconfiguration takes place. Regarding the measured metrics, we are observing both the times to launch a VM and to deliver it to application.

4.3.1 Performance and Elasticity Viability

Table 4 presents the results when executing the application over the aforementioned scenarios. The initial configuration starts from 2 or 4 nodes, with each node having two processors. Therefore, 4 and 8 VMs are launched using 2 and 4 nodes, respectively. Figures 8 and 9 depict the application's behavior with and without elasticity support, where the x-axis expresses the total time to execute the application.

TABLE 4
Elasticity results in two scenarios. Here, A, D and W mean the Ascending, Descending and Wave load patterns. The times are expressed in seconds.

Starting VMs	Load	Scenario i: AutoElastic with elasticity support				Scenario ii: AutoElastic without elasticity support			
		Time	Observations	Resource	Cost	Time	Observations	Resource	Cost
4	A	1978	65	26*4+27*6+12*8 = 362	716036	2426	81	81*4 = 324	786024
	D	1775	56	4*4+16*6+36*8 = 400	710000	2397	81	81*4 = 324	776628
	W	1895	59	4*4+22*6+33*8 = 412	780740	2444	82	82*4 = 328	801632
8	A	1789	57	39*6+18*8 = 378	676242	1555	52	52*8 = 416	646880
	D	1570	52	8*6+44*8 = 400	628000	1561	52	52*8 = 416	649376
	W	1686	55	19*6+36*8 = 402	677772	1599	53	53*8 = 424	677976

The results with the Constant load function were the same in both scenarios because AutoElastic does not suggest any allocation or deallocation during the application execution. Consequently, 2360 s and 1467 s were achieved when testing 4 and 8 VMs for the initial situation. Specifically, as presented in Table 4, the use of 4 VMs at the program launch implies better results when comparing scenarios i and ii. Figure 8 (a) shows two moments of VM allocation when working

with the Ascending function, providing a gain of 19% in the application time. The Descending load function achieves better results than the Ascending load function because the higher load in the beginning of the application forces two allocation actions sooner (see Figure 8 (b)). This configuration achieved a gain of approximately 26% when comparing AutoElastic with and without elasticity support. The Wave load function, on the other hand, presents the AutoElastic capacity to manage on-the-fly the number of VMs in accordance with the system load. Figure 8 (c) shows that when the load passes the limits of a threshold, an elasticity action is not triggered automatically because of the Aging concept that is employed by AutoElastic. Concerning the time perspective, elasticity provided gains of up to 22% when using this load function.

As a common feature in the load functions of Figure 8, the scenario without elasticity support passes more than half of the execution above the threshold of 80% of CPU usage. At the same time, this configuration presents peaks of up to 89% of the load when considering the total amount of available CPU. On the other hand, the architecture is not saturated when using AutoElastic capabilities. In addition to the gains in the time perspective when starting with 4 VMs, AutoElastic also presents better values for the decision function. The gain in time is not cost prohibitive; thus, elasticity is useful in this context both in terms of the performance and pay-as-you-go relation.

Figure 9 illustrates the results with 8 VMs in the starting configuration, running the parallel application. Considering part (a) of this figure, AutoElastic starts reducing the number of VMs, remains using 6 VMs during 1170 s and completes the execution again with 8 VMs in the most intensive phase of the application. As presented in Figure 9 (b), in the largest part of the execution, the used load remains between thresholds 640 and 320, which indicates that there are no elasticity actions. However, the reduction in the load occurs tenuously, which implies that the application takes a long time to reach the minimum threshold. Consequently, there is a waste of resources, mainly after crossing 900 s. In this context, a higher value for the minimum threshold could deallocate resources quickly, optimizing the *Resource* as presented in Equation 6. However, adaptable thresholds are not in the scope of this article and are being considered as future work. In addition, in contrast to Figure 8, the start with 8 VMs and without addressing the elasticity implies not crossing the maximum threshold of 80% or a 640 total CPU load. Thus, the execution without AutoElastic capabilities presents moments of underutilization of resources, but it is not possible to declare that the nodes were saturated at any time during the execution.

Despite not presenting good results in terms of the execution time, AutoElastic with 8 VMs at the beginning of the application shows encouraging results when examining the decision function. In this respect, AutoElastic obtained a better mark on two out of three values of the decision function. Auto-adjusting the values of the thresholds (fixed at 80% and 40%) should help to control the waste of resources and to boost more reactive (de)allocation actions during the load observations, reaching a better tuning of the decision function (to reduce the applications time using as small an amount of resources as possible). This work is not in the scope of this article and remains as future research for the next version of AutoElastic.

4.3.2 Analyzing Asynchronous Elasticity

Table 5 shows the times that are involved in the VM allocation actions. After analyzing the need for allocation, AutoElastic instantiates a new node (which is composed of two processors) and two instantiated virtual machines. This table presents 9 moments at which the parallel architecture goes up. Six observations is the result of the average between observing the need and actually delivering the VMs for the parallel application. The template to create a VM for the slave process occupies 700 Mbytes in memory; hence, 1.4 Gbyte is transferred at each allocation action. On average, AutoElastic spends 3 min and 36 s to instantiate the virtual machines. The computation (from the applications perspective) and communication (from the VM transferring viewpoint) occur in parallel, near the cloud computing and HPC panoramas. In previous work we surveyed many elasticity initiatives and showed that the average time to launch a VM varies from 1 to 10 minutes, so AutoElastic's time is enclosed on such an interval [9]. More precisely, we can define as competitive the AutoElastic's time to completely deliver a new VM, where [3] and [6] showed values close to 10 minutes, while Marshall et al. [24] obtained 3 min and half for this procedure.

TABLE 5
Analyzing the the time interval between detecting the need to allocate and the delivery of 2 VMs at each elasticity action. The times are expressed in seconds.

Starting VMs	Load Function	Index of the observation		Instant of Time		Time interval to deliver the VMs
		Allocation of VMs	Delivering of VMs	Allocation of VMs	Delivering of VMs	
4	Ascending	26	32	752	958	206
		52	58	1557	1764	207
	Descending	4	10	90	295	205
		13	20	388	625	237
	Wave	4	10	91	297	206
		13	19	390	597	207
		38	45	1211	1447	236
8	Ascending	42	49	1283	1524	241
	Wave	34	40	1007	1214	207
					Average	216

Figure 10 depicts the impact of the connection and disconnection of processes on running slave processes. Each graph presents two CPUs, each one running a single VM in which only one slave process is actually running. Figure 10 (a) shows that the impact of disconnection is almost null because the master process closes only the sockets that have the

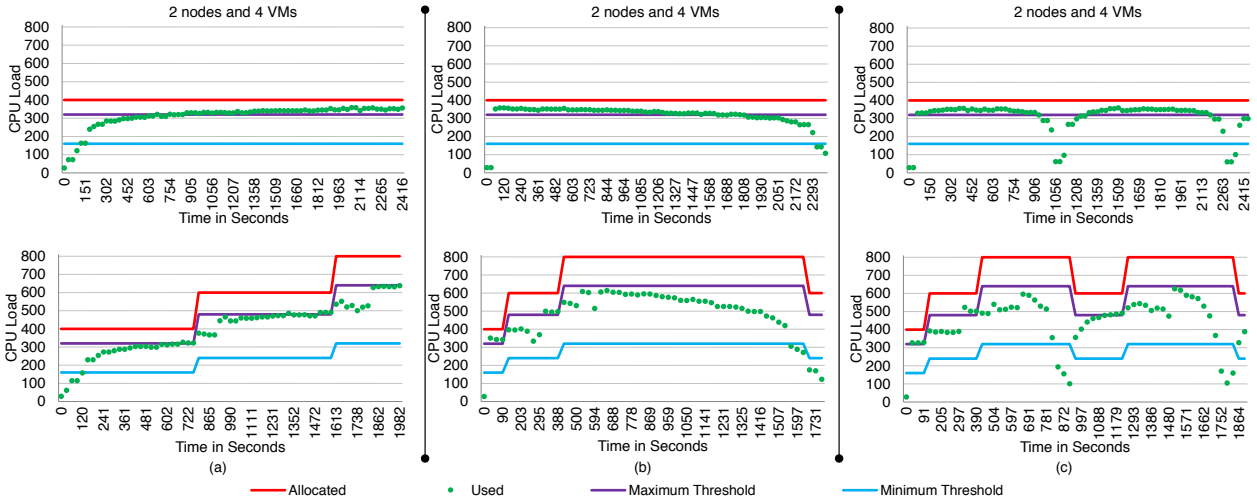

Fig. 8. CPU behavior when starting with 2 nodes and the following load patterns: (a) Ascending; (b) Descending; (c) Wave. Non-elastic and elastic executions are expressed in the upper and the bottom parts, respectively.

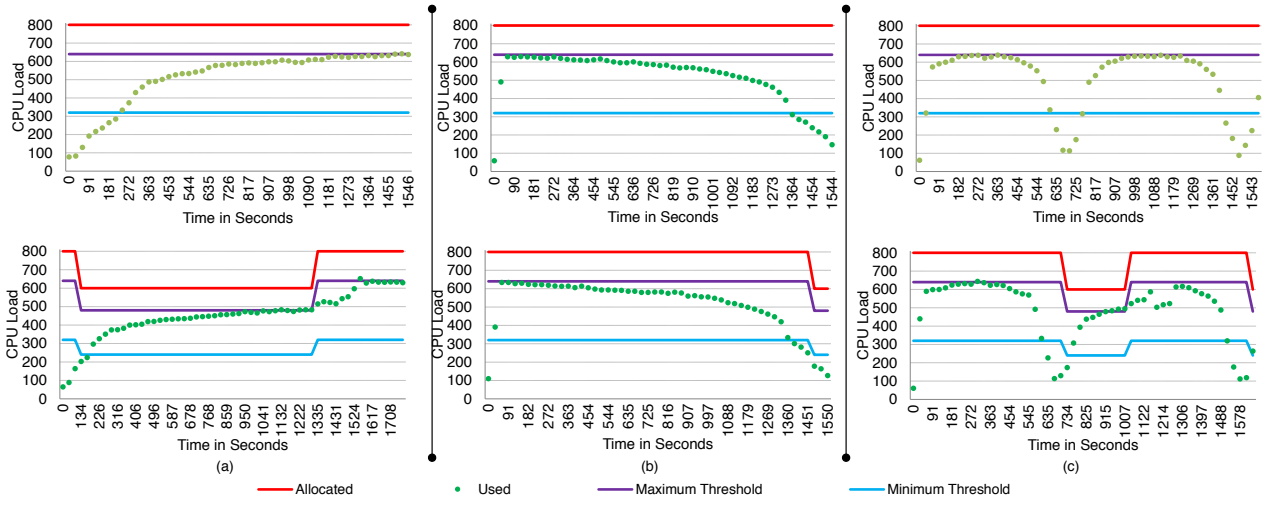

Fig. 9. CPU behavior when starting with 4 nodes and the following load patterns: (a) Ascending; (b) Descending; (c) Wave. Non-elastic execution is in the upper part, while elasticity capacity appears in the bottom.

processes that belong to the VMs that will be deallocated afterward. This part of the figure also shows a growing tendency toward CPU consumption because now fewer VMs must take on the application workload. Figure 10 (b) presents a clear gap in the CPU utilization of two running processes. In the allocation action, the master must establish two new connections synchronously. Only after completing this task, the master obtains the roll of current processes and can pass a uniform part of the work for each process. Moreover, it is possible to observe a short reduction in the CPU usage after crossing 1770 s because of the workload spread among more processes after an allocation action.

5 Conclusion

Cloud elasticity has been used for scaling traditional web applications to handle unpredictable workloads, which enables companies to avoid downfalls that are involved with fixed provisioning (over- and under-provisioning). Considering the current scenario, this article addressed cloud elasticity for HPC applications by presenting a model denoted AutoElastic. In brief, AutoElastic self-organizes the number of virtual machines without any user intervention, providing benefits for both the cloud administrator (better energy consumption and resource sharing among the users) and the cloud user (application performance and overhead when migrating to the cloud). As the main scientific contribution to the state-of-the-art in the confluence of cloud and HPC systems, AutoElastic proposes the use of asynchronism for elasticity management. This feature implies not blocking the parallel application while a newly allocated VM is being transferred, which offers one of the most important capabilities of the cloud without prohibitive costs.

The AutoElastic's model for applications matches the socket-programming style that is offered by MPI-2, in which new processes can be easily added or re-

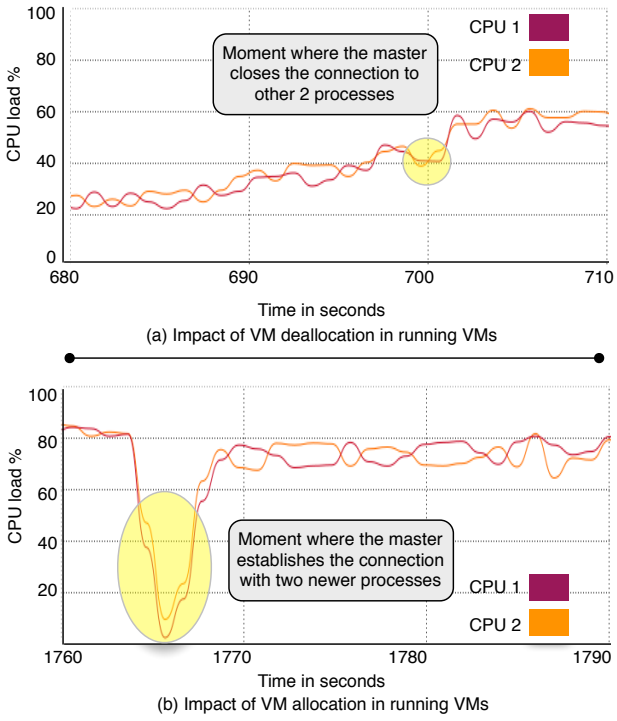

Fig. 10. Impact in the running processes regarding the disconnection and connection of other processes of the parallel application: (a) start with 4 VMs and Wave function; (b) start with 4 VMs and Ascending function

moved, connected and disconnected, from the parallel computation. The evaluation demonstrated encouraging results on using the CPU metric for HPC elasticity. Despite addressing a processing-intensive application, they revealed that the load does not always remain close to 100%. We measured that the overhead to double the VMs is on average 3 min and 36 s, but during this interval, the application continues to execute normally. Furthermore, the evaluation presented a decision function that is expressed by multiplying the resource consumption by the performance, where AutoElastic achieved 5 positive results on 6 observations when compared with a non-elastic execution.

As future work, we concern the study of network and memory elasticity to employ these capabilities in the next versions of AutoElastic. We also plan to extend AutoElastic to cover divide-and-conquer and bulk-synchronous parallel applications.

Acknowledgment

The authors would like to thank to the following Brazilian agencies: CNPq, CAPES and FAPERGS.

References

[1] W. Lin, J. Z. Wang, C. Liang, and D. Qi, "A threshold-based dynamic resource allocation scheme for cloud computing," *Procedia Engineering*, vol. 23, no. 0, pp. 695 – 703, 2011.

[2] J. Duggan and M. Stonebraker, "Incremental elasticity for array databases," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM, pp. 409–420, 2014.

[3] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in *Proc. of the 9th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, New York, NY, USA: ACM, pp. 95–104, 2014.

[4] W. Dawoud, I. Takouna, and C. Meinel, "Elastic VM for cloud resources provisioning optimization," in *Advances in Computing and Communications*, ser. Comm. in Comp. and Information Science, Springer Berlin, vol. 190, pp. 431–445, 2011.

[5] S. Imai, T. Chestna, and C. A. Varela, "Elastic scalable cloud computing using application-level migration," in *Proc. of the Int. Conf. on Utility and Cloud Computing*, Washington, DC, USA: IEEE Computer Society, pp. 91–98, 2012.

[6] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Grid Computing (GRID), 11th IEEE/ACM Int. Conf. on*, pp. 41 –48, 2010.

[7] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," *Cluster Computing and the Grid, IEEE Int. Symp. on*, vol. 0, pp. 644–651, 2012.

[8] A. Raveendran, T. Bicer, and G. Agrawal, "A framework for elastic execution of existing MPI programs," in *Proc, of the Int. Symp. on Parallel and Dist. Processing Workshops and PhD Forum*, Washington, DC, USA: IEEE Comp. Society, pp. 940–947, 2011.

[9] G. Galante and L. C. E. d. Bona, "A survey on cloud computing elasticity," in *Proc. of the Int. Conf. on Utility and Cloud Computing*, Washington: IEEE Comp. Soc., pp. 263–270, 2012.

[10] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 7:1–7:47, 2014.

[11] D. Chiu and G. Agrawal, "Evaluating caching and storage options on the Amazon Web Services cloud," in *Grid Computing (GRID), 2010 11th IEEE/ACM Int. Conf. on*, pp. 17 –24, 2010.

[12] E. Lusk, "MPI-2: standards beyond the message-passing model," in *Massively Parallel Programming Models, Proc.. Third Working Conf. on*, pp. 43–49, 1997

[13] D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain, "Converting a high performance application to an elastic cloud application," in *Proc. of the Conf. on Cloud Computing Technology and Science*, Washington, USA: IEEE Comp. Soc., pp. 383–390, 2011.

[14] T. Knauth and C. Fetzer, "Scaling non-elastic applications using virtual machines," in *Cloud Computing (CLOUD), IEEE Int. Conf. on*, pp. 468 –475, 2011.

[15] K. Kumar, J. Feng, Y. Nimmagadda, and Y.-H. Lu, "Resource allocation for real-time tasks using cloud computing," in *Comp. Comm. and Networks (ICCCN), Int. Conf. on*, pp. 1 –7, 2011.

[16] E. Michon, J. Gossa, and S. Genaud, "Free elasticity and free CPU power for scientific workloads on IaaS clouds," in *Parallel and Distributed Systems (ICPADS), Int. Conf. on*, pp. 85 –92, 2012.

[17] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. Wright, "Performance analysis of high performance computing applications on the Amazon Web Services cloud," in *Cloud Computing Technology and Science (CloudCom), Int. Conf. on*, pp. 159–168, 2010.

[18] B. Wilkinson and C. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Pearson/Prentice Hall, 2005.

[19] A. Tanenbaum, *Computer Networks*, 4th ed. Upper Saddle River, New Jersey: Prentice Hall PTR, 2003.

[20] D. Milojicic, I. M. Llorente, and R. S. Montero, "Opennebula: A cloud management tool," *Internet Computing, IEEE*, vol. 15, no. 2, pp. 11 –14, march-april 2011.

[21] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proc. of the 2011 Int. Conf. on Distributed Computing Systems*, Washington, DC, USA: IEEE Computer Society, pp. 559–570, 2011.

[22] A. Lonea, D. Popescu, and O. Prostean, "A survey of management interfaces for Eucalyptus cloud," in *Applied Comp. Intellig. and Informatics (SACI), IEEE Symp.*, pp. 261–266, 2012.

[23] E. Roloff, F. Birck, M. Diener, A. Carissimi, and P. Navaux, "Evaluating high performance computing on the Windows Azure platform," in *Cloud Computing (CLOUD), 2012 IEEE 5th Int. Conf. on*, pp. 803 –810, 2012.

[24] P. Marshall, K. Keahey, and T. Freeman, "Elastic Site: Using clouds to elastically extend site resources," in *Proc. of the IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, Washington, DC, USA: IEEE Comp. Society, pp. 43–52, 2010.

[25] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang, "Comparison of open-source cloud management platforms: Openstack

and Opennebula," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th Int. Conf. on*, pp. 2457 –2461, 2012.
[26] B. Cai, F. Xu, F. Ye, and W. Zhou, "Research and application of migrating legacy systems to the private cloud platform with Cloudstack," in *Automation and Logistics (ICAL), IEEE Int. Conf. on*, pp. 400 –404, 2012.
[27] L. Surhone, M. Tennoe, and S. Henssonow, *Rightscale*. VDM Publishing, 2010.
[28] T. Fujii and M. Kimura, "Analysis results on productivity variation in Force.com applications," in *6th Int. Conf. on Software Process and Product Measurement (IWSM)*, pp. 314 –317, 2011.
[29] L. Fu and C. Gondi, "Cloud computing hosting," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE Int. Conf. on*, vol. 3, pp. 194–198, 2010.
[30] R. Costa, F. Brasileiro, G. L. de Souza Filho, and D. M. Sousa, "Just in Time Clouds: Enabling Highly-Elastic Public Clouds over Low Scale Amortized Resources," Universidade Federal de Campina Grande, Tech. Rep., 2010.
[31] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proc. of the 2nd ACM Symp. on Cloud Computing*, New York, NY, USA: ACM, pp. 5:1–5:14, 2011.
[32] P. Martin, A. Brown, W. Powley, and J. L. Vazquez-Poletti, "Autonomic management of elastic services in the cloud," in *Proc. of the IEEE Symp. on Computers and Communications*, Washington, DC, USA: IEEE Comp. Society, pp. 135–140, 2011.
[33] I. Moreno and J. Xu, "Customer-aware resource overallocation to improve energy efficiency in realtime cloud computing data centers," in *Service-Oriented Computing and Applications (SOCA), 2011 IEEE Int. Conf. on*, dec. pp. 1 –8, 2011.
[34] X. Zhang, Z.-Y. Shae, S. Zheng, and H. Jamjoom, "Virtual machine migration in an over-committed cloud," in *Network Operat. and Management Symp. (NOMS)*, pp. 196 –203, 2012.
[35] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proc. of the 4th USENIX conf. on Networked systems desig*, Berkeley, CA, USA: USENIX Association, pp. 17–23, 2007.
[36] L. Beernaert, M. Matos, R. Vilaça, and R. Oliveira, "Automatic elasticity in openstack," in *Proc. of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, New York, NY, USA: ACM, pp. 2:1–2:6, 2012.
[37] N.-L. Tran, S. Skhiri, and E. Zimányi, "EQS: An elastic and scalable message queue for the cloud," in *Proceedings of the 2011 IEEE Third Int. Conf. on Cloud Computing Technology and Science*, Washington, DC, USA: IEEE, pp. 391–398, 2011.
[38] B. Suleiman, "Elasticity economics of cloud-based applications," in *Proc. of the Int. Conf. on Services Computing*, Washington, DC, USA: IEEE Computer Society, pp. 694–695, 2012.
[39] X. Zhang, S. Jeong, A. Kunjithapatham, and S. Gibbs, "Towards an elastic application model for augmenting computing capabilities of mobile platforms," in *Mobile Wireless Middleware, Operating Systems, and Applications*, ser. Lecture Notes of the Institute for Computer Sciences. Springer Berlin Heidelberg, vol. 48, pp. 161–174, 2010.
[40] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: cloud micro-elasticity via VM state coloring," in *Conf. Computer Systems*, ser. EuroSys '11. NY, USA: ACM, pp. 273–286, 2011.
[41] D. Kumar, Z.-Y. Shae, and H. Jamjoom, "Scheduling batch and heterogeneous jobs with runtime elasticity in a parallel processing environment," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th Int.*, pp. 65 –78, 2012.
[42] S. Mariani, H.-L. Truong, G. Copil, A. Omicini, and S. Dustdar, "Coordination-aware elasticity," in *7th IEEE/ACM Int. Conf. on Utility and Cloud Comp. (UCC)*, London, pp. 56–63, 2014.
[43] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith, "Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation," in *Proc. of the Int. Conf. on Self-Adaptive and Self-Org. Systems (SASO)*, 2014.
[44] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovic, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *Computer Architecture (ISCA), 2011 38th Annual Int. Symp. on*, pp. 129–140, 2011.
[45] J. Baliga, R. Ayre, K. Hinton, and R. Tucker, "Green cloud computing: Balancing energy in processing, storage, and transport," *Proc. of the IEEE*, vol. 99, no. 1, pp. 149–167, 2011.
[46] M. Comanescu, "Implementation of time-varying observers used in direct field orientation of motor drives by trapezoidal integration," in *Power Electronics, Machines and Drives (PEMD 2012), 6th IET Int. Conf. on*, pp. 1–6, 2012.

Rodrigo da Rosa Righi is assistant professor and researcher at University of Vale do Rio dos Sinos, Brazil. Rodrigo concluded his post-doctoral studies at KAIST - Korea Advanced Institute of Science and Technology, under the following topics: RFID and cloud computing. He obtained his PhD degree in Computer Science from the UFRGS University, Brazil, in 2005. His research interests include load balancing and process migration. Finally, he is a member of the IEEE and ACM.

Vinicius Facco Rodrigues completed his Bachelors degree in Computer Science at Unisinos University in 2012. In addition, he started his masters in Applied Computing at the same University in 2014. His research areas include distributed systems and computer networks. Currently, Vinicius is focusing his research on the topic of cloud computing and, more specifically, on the elasticity feature of this new paradigm.

Cristiano Andé da Costa is a full professor at Universidade do Vale do Rio dos Sinos (Unisinos), Brazil, and a researcher on productivity at CNPq (National Council for Scientific and Technological Development). His research interests include ubiquitous, mobile, parallel and distributed computing. He obtained his PhD degree in computer science from the UFRGS University, Brazil, in 2008. He is a member of the ACM, IEEE, IADIS and the Brazilian Computer Society (SBC).

Tiago Ferreto is Associate Professor of Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil. He received his Ph.D. in Computer Science from the Computer Science Department, PUCRS, Brazil (2010), with a Sandwich PhD Internship at TU-Berlin, Germany, in 2008. He had several publications in prestigious conferences and journals. His research interests are cloud computing, IT infrastructure management and computer networks.

Guilherme Galante concluded his bachelors in Informatics at Universidade Estadual do Oeste do Parana in 2003. He obtained his masters at Universidade Federal do Rio Grande do Sul, 2006, and his PhD at Universidade Federal do Parana, 2014. Since 2006, Guilherme has acted as professor in the Computing Science undergraduate course at Universidade Estadual do Oeste do Parana. His research areas include computational systems and applied computing.

Luis Carlos Erpen de Bona is an adjunct professor at Universidade Federal do Parana. He obtained both bachelors in (1999) and Masters (2002) degrees in Informatics at the same university. In addition, he concluded his PhD in Electric Engineering at UTFPR. He has experience in computer science, especially in distributed systems. He acted as coordinator of several research technological development projects, for both national and international perspectives.