CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3710

Joint-analysis of performance and energy consumption when enabling cloud elasticity for synchronous HPC applications

Rodrigo da Rosa Righi*,†, Cristiano André da Costa, Vinicius Facco Rodrigues and Gustavo Rostirolla

Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos (Unisinos), São Leopoldo, RS, Brazil

SUMMARY

A key characteristic of cloud computing is elasticity, automatically adjusting system resources to an application's workload. Both reactive and horizontal approaches represent traditional means to offer this capability, in which rule-condition-action statements and upper and lower thresholds occur to instantiate or consolidate compute nodes and virtual machines. Although elasticity can be beneficial for many HPC (high-performance computing) scenarios, it also imposes significant challenges in the development of applications. In addition to issues related to how we can incorporate this new feature in such applications, there is a problem associated with the performance and resource pair and, consequently, with energy consumption. Further exploring this last difficulty, we must be capable of analyzing elasticity effectiveness as a function of employed thresholds with clear metrics to compare elastic and non-elastic executions properly. In this context, this article explores elasticity metrics in two ways: (i) the use of a cost function that combines application time with different energy models; (ii) the extension of speedup and efficiency metrics, commonly used to evaluate parallel systems, to cover cloud elasticity. To accomplish (i) and (ii), we developed an elasticity model known as AutoElastic, which reorganizes resources automatically across synchronous parallel applications. The results, obtained with the AutoElastic prototype using the OpenNebula middleware, are encouraging. Considering a CPU-bound application, an upper threshold close to 70% was the best option for obtaining good performance with a non-prohibitive elasticity cost. In addition, the value of 90% for this threshold was the best option when we plan an efficiency-driven execution. Copyright © 2015 John Wiley & Sons, Ltd.

Received 14 April 2015; Revised 2 September 2015; Accepted 15 September 2015

KEY WORDS: cloud computing; elasticity; energy consumption; metrics; elastic speedup; elastic efficiency

1. INTRODUCTION

A key feature of cloud computing is elasticity, in which users can scale their computational resources up or down at any moment according to demand or desired response time [1,2]. Considering the high-performance computing (HPC) landscape and a very long-running parallel application, a user may want to increase the number of instances to try to reduce the completion time of the application. Logically, the success of this procedure will depend on both the computational grain and application modeling [3,4]. Conversely, if an application is not scaling in a linear or close to linear fashion and if the user is flexible with respect to completion time, the number of instances can be reduced. This results in a lower nodes × hours index, and thus in a lower cost and better energy saving. Because of performance advances in the virtualization thematic [5], cloud elasticity could appear as a viable alternative to yield significant cost savings for HPC users when compared with the traditional approach of maintaining an expensive cluster-based IT infrastructure [6]. Normally, in this last case,

*Correspondence to: Rodrigo da Rosa Righi, Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos (Unisinos), 93.022–000, São Leopoldo, RS, Brazil.
† E-mail: rrrighi@unisinos.br

there is a provision for peak usage [7], which is underutilized when we observe the entire application execution or when analyzing the real use of the infrastructure.

Today, the combination of horizontal and reactive approaches represents the most used methodology for delivering cloud elasticity [2,4,8,9]. In this approach, rule-condition-action statements with upper and lower load thresholds occur to drive the allocation and consolidation of instances, either nodes or virtual machines (VMs). Despite being simple and intuitive, this method requires programmer's technical skills for tuning the parameters. Moreover, these parameters can vary according to the application and the infrastructure. Specifically for the HPC panorama, the upper threshold defines for how long the parallel application should run close to the overloaded state. For example, a value close to 75% implies always executing the HPC code in a non-saturated environment, but paying for as many instances (and increasing energy consumption) as needed to reach this situation. Conversely, a value near 100% for this threshold is relevant for energy saving when analyzing the number of allocations. However, this can result in weaker application reactivity, which can generate penalties in execution time. The lower threshold, in turn, is useful for deallocating resources, whereas a value close to 0% will delay this action. This scenario can cause overestimation of resource usage because the application will postpone consolidations [10]. The main reason for that behavior in HPC and dynamic applications is to avoid VM deallocations because, in the near term, the application could increase its need for CPU capacity, forcing another time-costly allocation action.

In HPC systems, elasticity can be a double-edged sword involving performance and energy consumption. Directly related to both, we have resource consumption, which can also help in measuring elasticity quality. Although elasticity allows applications to acquire and release resources dynamically, adjusting to changing demands, deciding about appropriate thresholds and measuring performance and energy consumption accurately are not easy tasks [1,2]. Our analysis shows that related work presents a gap concerning defining decision functions for elasticity viability in the combination of performance and energy, besides not addressing a discussion of lower and upper thresholds [2,4,6,11–28]. Responding to this motivation, we developed an elasticity model called AutoElastic,[‡] which automatically reorganizes resources for loop-based synchronous parallel applications. AutoElastic acts at the platform as a service (PaaS) level of a cloud, hiding any details from users about horizontal elasticity in terms of both application writing and threshold management.

Thus, this article presents AutoElastic fundamentals, focusing primarily on both performance and energy consumption topics in the HPC and cloud elasticity pair. More precisely, we address this pair through two research questions: (i) considering a CPU-bound application and different application workloads, what are the best threshold parameters for time, energy and cost? and (ii) what is the relationship between predicted energy consumption and resource allocations? To answer these questions, we perform an analysis of a set of lower-upper thresholds and different computational workloads (ascending, descending, constant, and wave) across a CPU-intensive HPC application, which was compiled together with the AutoElastic prototype. In addition, we propose novel metrics to evaluate the aforementioned pair. Thus, the article's scientific contributions are threefold, as stated:

(i) An energy model that consists of a framework to measure and predict energy consumption in elastic cloud environments comprising homogeneous compute nodes;
(ii) Comparison of synergies of two cost functions, the first based on energy consumption and the second based on resource use, considering both elastic and non-elastic environments;
(iii) Redefinition of the so-called speedup and parallel efficiency metrics for elastic HPC environments, here denoted as elastic speedup (ES) and elastic efficiency (EE).

Our understanding is that these contributions are a basis for discussing the convergence of cloud and HPC. The results, when executed using the OpenNebula [29] private cloud, reveal the importance of tuning the thresholds to enable elasticity in a non-prohibitive way. Specifically, the results present the relationship among EE, ES and cost, and thresholds (lower, upper or combination of both) for each type of considered workload. The remainder of this article will first introduce related works in Section 2. Section 3 discusses AutoElastic and the aforementioned contributions in detail.

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

DOI: 10.1002/cpe

The evaluation methodology and a discussion of the results are presented in Sections 4 and 5. Finally, Section 6 presents the final remarks, highlighting the contributions with quantitative data and presenting directions for future work.

2. RELATED WORK

This article describes a cloud elasticity initiative called AutoElastic, which is evaluated using the proposed metrics. Thus, we organized this section in two groups: (i) performance and/or energy-driven cloud computing middleware and models [11–19]; (ii) metrics to measure energy consumption and cost in cloud environments [2,4,6,20–28]. These groups are addressed in Sections 2.1 and 2.2. Finally, we discuss research opportunities and open issues as a joint analysis of both previously mentioned groups in Section 2.3.

2.1. Cloud computing middleware and energy models

Some works focus on the trade-off between energy and performance when running HPC applications in the cloud. Tian *et al.* [11] address the problem of optimal dynamic speed scaling and job scheduling for multiple heterogeneous servers. The purpose is to optimize the performance and power consumption trade-off. To accomplish this, the authors define a model that receives a β value; a larger β means higher weight on power consumption and lower performance restriction, and vice versa. Paya and Marinescu [12] developed an energy-aware operation model for a server S that identifies an optimal operating region, two suboptimal, and two undesirable ones. The basic philosophy of their approach is to define a power-optimal operation region for each server and to attempt to maximize the number of servers operating in this region. In cloud computing, a critical goal is minimizing the cost of providing the service and, in particular, minimizing energy consumption. Scaling decisions are made at several levels: (i) local decision perform vertical scaling using local resources, with no need to interact with the leader; (ii) in-cluster, horizontal or vertical scaling migrate some of the applications to other servers identified by the leader; and (iii) inter-cluster scaling the leader determines which cluster does not have available capacity to respond to a request to increase its allocation.

According to Luo *et al.* [13], an energy-aware resource management algorithm must consider both energy consumption and quality of service requirements because in a cloud environment, VMs have an Service Level Agreement (SLA) to specify performance parameters such as CPU, memory, disk, and network. The article presents a model to predict energy consumption inside a single machine, in addition to a simulation-driven framework to evaluate energy-aware resource schedule algorithms. The authors claim that in most existing cloud computing energy studies, linear models are used to describe the relationship between energy consumption and resource utilization. More precisely, this project explores a variety of regression analysis methods to estimate energy consumption accurately with low computational overhead. Beloglazov *et al.* [14] propose an energy-efficient resource management system for virtualized cloud data centers that avoids violating SLAs by using live migration. CPU, RAM memory, and network bandwidth are used as metrics to drive VM reallocation. Local managers choose VMs that must be migrated if (i) the utilization of some resource is close to 100%, causing a risk to the SLA; (ii) the utilization of resources is low, requiring reallocation of the resources and turning off the node; (iii) intensive network communication occurs with a VM allocated in another node, suggesting migration of the VM to the other node if possible; or (iv) the temperature exceeds an established limit.

Feifei *et al.* [30] developed StressCloud, a performance and energy consumption profiling and analysis tool for cloud systems. Using StressCloud, they have conducted extensive experiments to analyze empirically the performance and energy consumption of cloud systems. Their experimental results demonstrate the relationship between the performance and energy consumption of cloud systems with different resource allocation strategies and workloads. However, they address neither parallel applications nor cloud elasticity. Fargo *et al.* [31] developed a technique named Autonomic performance-per-watt management. The central idea behind this work is to proactively predict changes in current workloads and dynamically adjust hardware resources (number of cores

DOI: 10.1002/cpe

and amount of memory) assigned to the virtual machines (VMs) that run these workloads while satisfying SLA requirements. The authors work with previous information about the tasks, including their number, type, and size. Chuang *et al.* [32] affirm that we need a model in which programmers do not need be aware of the actual scale of the application or of the runtime support that dynamically reconfigures the system to distribute application state across computing resources. To address this point, the authors developed EventWave, a programming model and runtime support for tightly coupled elastic cloud applications. The initiative focuses on a game server and works only with VM migration.

Wei *et al.* [15] explore elastic resource management on the PaaS level. When applications are deployed, they are first allocated on separate servers, so that they can be monitored more meticulously. Then, the authors collect long-term monitoring data to estimate the application's characteristics, time that is added to the normal execution of the application. Leite *et al.* [16] developed a middleware named Excalibur to execute parallel applications in the cloud automatically. Excalibur works with independent tasks organized in different partitions. The algorithms must to know some information in advance, such as the total number of tasks and the estimated CPU time to execute each type of task. Al-Shishtawy and Vlassov [17] present the design and evaluation of ElastMan, an elasticity manager for key-value applications in the cloud. ElastMan automatically resizes an elastic service in response to changes in workload to meet SLAs at a reduced cost. By combining feed-forward control and feedback control, ElastMan addresses the challenges of the variable performance of cloud VMs. Aniello *et al.* [18] developed an architecture for automatically scaling replicated services. A queuing model of the replicated service is used to compute the expected response time, given the current configuration (number of replicas) and the distributions of both input requests and service times. However, a queuing system is not the best option for dynamic applications and/or infrastructure because it is based on rigid parameters. Gutierrez-Garcia and Sim [19] developed an agent-based framework to address cloud elasticity for bag-of-tasks demands. The status of the BoT execution is verified every hour; if it is not completely executed, all the cloud resources are reallocated for the next hour.

2.2. *Cloud computing metrics to analyze performance and efficiency*

This section presents some metrics to evaluate cloud systems and applications. Roloff *et al.* [6] define a cost efficiency metric, which is obtained by considering the average performance and financial cost per hour when running HPC applications in the cloud. This metric can be used to measure the cost efficiency of different platforms. The obtained results emphasize that cloud platforms provide a viable alternative when compared with maintaining an in-house cluster. Tesfatsion *et al.* [20] perform a joint analysis of cost and energy performance using techniques such as dynamic voltage and frequency scaling, horizontal, and vertical elasticity. This combined approach resulted in 34% energy saving compared with when each policy is applied alone. The authors also define a monetary cost function that maps the performance penalty through the trade-off between energy efficiency and performance. Martin *et al.* [21] provide two metrics to analyze a system's behavior: $\frac{cost}{performance}$ and $\frac{cost}{throughput}$. In their work, cost refers to the financial cost of executing an application in the cloud, performance means the makespan to compute a series of tasks, and throughput is the number of tasks completed in accordance with time.

Coutinho *et al.* [22] explore cost/performance and cost/bandwidth rates. In addition to these metrics, they also propose a metric, hours/instances, that contemplates the total execution time and the number of instances required. Weber *et al.* [2] present two metrics to analyze cloud elasticity. The idea consists of comparing the number of scale up (or scale down) events D_u (or D_d) of the demand with the number of scale up (or scale down) events A_u (or A_d) for allocated resources. Thus, they compute *scale* as $\frac{|du-au|}{du}$, in which the intent is to obtain a value close to 0. Too many scale events for the resource supply and too few scale events are both indicators of bad timing behavior. In addition, they measure another metric to evaluate the efficiency of resource usage. They claim that efficiency is defined by $E = \frac{1}{(A.U)}$, in which A is the average time to switch from an under-provisioned state to an optimal or over-provisioned state and corresponds to the average speed of scaling up, whereas U refers to the average amount of under-provisioned resources during an under-provisioned period.

DOI: 10.1002/cpe

Islam *et al.* [23] provide a quantitative definition of elasticity using financial terms, adopting the point of view of an elastic system customer who wants to measure the elasticity provided by the system. The authors compute the financial penalty for systems' under-provisioning (owing to SLA violations) and over-provisioning (unnecessary costs) using a reference benchmark suite to characterize system elasticity. They develop a model to measure a penalty to a consumer for under-provisioning (leading to unacceptable latency or unmet demand) or over-provisioning (paying more than necessary for the resources needed to support a workload). Technically, the penalty model computes numerical integration considering resource demand and supply. To assess sensitivity to CPU usage, Tembey *et al.* [24] use the following metric: CPU utilization divided by allocated CPU. Wang *et al.* [25] use the standard cost function from Amazon: $Cost = Price \times Time$. Here, *Time* is the total running time of the workload in hours, and *Price* is the price per VM hour. Similarly, Garg *et al.* [26] present the following cost equation: $cost = Energy \times Time$. In [27], the authors use the following equation to measure energy: $E = P \times Time$. The amount of energy used depends on the power and the time for which it is used. Thus, E, P, and *Time* denote energy, power, and time, respectively. The standard unit for energy is the joule (J), assuming that power is measured in watts (W) and time in seconds (s).

Jamshidi *et al.* [4] and Bersani *et al.* [28] work on defining cloud elasticity terms. Jamshidi *et al.* [4] define three terms for elasticity performance: (i) scalability, the ability of the system to sustain workload fluctuations; (ii) cost efficiency, acquiring only required resources by releasing underutilized ones; and (iii) time efficiency, acquiring and releasing resources as soon as a request is made. Bersani *et al.* [28] define eagerness, precision, and resource thrashing. Eagerness informally refers to the reaction speed of a system upon a change in the load. Precision identifies how good is the elastic system at allocating and deallocating the right number of resources with respect to variation in the load. Finally, resource thrashing refers to oscillation in allocation and deallocation actions. In other words, elastic systems may present opposite adaptations in a very short time, for example, a system may scale up, and then, right after finishing the adaptation, it can start to scale down. This situation is commonly known as resource thrashing.

2.3. Research opportunities

Observing the two previous sections, we envision gaps to address cloud elasticity for HPC environments in both cloud middleware and metric levels. First, our study does not find an initiative that concomitantly addresses transparency, tightly coupled HPC applications, and VM allocation with thrashing avoidance when providing the cloud elasticity feature. Here, transparency is applicable to the user (who commonly is committed to configuring elasticity parameters such as thresholds, rules or actions), knowledge (referring to data that must be provided in advance at compilation or development to feed the elasticity engine), and application writing (the insertion of monitoring and/or elasticity directives directly in the application code). In addition, the term thrashing refers to consecutive oscillations on allocation and deallocation actions, largely when crossing an application peak or drop of performance.

Considering the metrics perspective, we observed that the most common technique includes the ratio of cost by performance. Cost divided by throughput or bandwidth, hours divided by instances, and the relationship of CPU allocated against CPU used are metrics already presented in related work. Moreover, considering the cost, we observe its computation by using either time and energy or time and price metrics. Concerning energy consumption, the traditional method that considers power and time is usually employed. Thus, in addition to the aforementioned problem at the middleware level, we highlight the following concerning evaluation metrics: (i) energy consumption evaluation considers a malleable number of resources; (ii) in elastic environments, there is a lack of joint analysis of energy consumption and resource usage when deciding the values for lower and upper elasticity thresholds; and (iii) common metrics, such as speedup and efficiency, are used to evaluate parallel programs in cloud computing, with a fixed number of resources, but not on elastic infrastructures. AutoElastic works to offer alternatives to fill the previously mentioned gaps. In the next section, we present the model in detail.

DOI: 10.1002/cpe

3. AUTOELASTIC – AN ELASTICITY MODEL FOR HPC APPLICATIONS

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 (in which tools such as PBS,[§] OAR,[¶] and OGS[‖] are usually employed for resource reservation and job scheduling) and energy consumption. In addition, the tuning of the number of processes to execute an HPC application can be a difficult procedure: (i) both short and large values will not efficiently explore the distributed system and (ii) a fixed value cannot fit irregular applications, in which the workload varies during execution and/or occasionally is not predictable in advance. Conversely, cloud elasticity abstracts the infrastructure configuration and technical details about resource scheduling from users, who pay for resources, and consequently energy, in accordance with applications' demands. However, the main gaps in the HPC and elasticity pair are application modeling and the overhead related to scaling operations. Aiming at addressing these gaps, we propose AutoElastic – a cloud elasticity model that operates at the PaaS level of a cloud, acting as middleware that enables the transformation of a non-elastic parallel application into an elastic one.

AutoElastic works with both horizontal and reactive elasticity [9], providing allocation and consolidation of compute nodes and VMs that run an iterative parallel application. This type of construction was adopted because the most successful MPI programs are written in a bulk synchronous style in which computational processes proceed in phases [33]. AutoElastic provides reactive elasticity transparently and effortlessly to the user, who does not need either to write elasticity rules and actions or to manage thresholds for resource reconfiguration. In addition, users need not change their parallel application, either by inserting any elasticity calls from a particular cloud library or by modifying the application code to add/remove resources by themselves.

3.1. Architecture

Figure 1(a) depicts the AutoElastic's fundamentals, emphasizing the roles of a manager, denoted AutoElastic manager, and of the user, who needs only to care about his or her parallel application. Figure 1(b) illustrates the AutoElastic's architecture, exploring the interactions among the application, the resources, the cloud front-end, and the AutoElastic manager. The architecture contemplates m homogeneous and distributed computational nodes, of which at any given time at least one is active. This node is in charge of running a VM with the master process and c other VMs with slave processes (one slave per VM), in which c equals the number of processing units (cores or CPUs) inside a particular node. This approach is based on the work of Lee *et al.* [34], which seeks to explore a better efficiency in parallel applications. The elasticity grain for each scaling in or out action refers to a single node and, consequently, to its VMs and processes. Finally, at any time, the number of VMs running slave processes is equal to $n = c \times m$.

AutoElastic is aware of the overhead to instantiate a VM, offering a framework that includes a manager that enables a feature named asynchronous elasticity. This feature was modeled specifically to work with master-slave loop-based parallel applications, in which the manager controls scaling operations concomitantly with the application execution. The communication between the manager and the master process is performed using a shared memory area, which can be enabled with network file system (NFS), Advanced Message Queuing Protocol (AMQP), or JavaSpaces. The use of a shared area for data interaction among VM instances is a common approach in private clouds [29,35,36]. At each loop iteration, the master verifies whether a manager signalizes the existence of resources to add or drop to/from the communication topology. Specifically, AutoElastic enables three actions, as discussed in the following:

- The manager writes to the shared area, whereas application processes read from it:

[§]Project website: http://www.arc.ox.ac.uk/content/pbs.
[¶]Project website: https://oar.imag.fr/.
[‖]Previously known as Sun Grid Engine. Project website: http://gridscheduler.sourceforge.net.

DOI: 10.1002/cpe

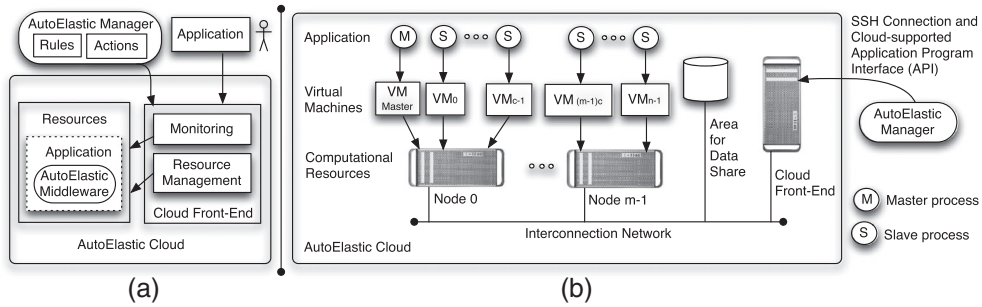


Figure 1. (a) AutoElastic ideas, emphasizing the roles of a manager and the user, who need only care about his or her application; (b) AutoElastic architecture. Here, c denotes the number of cores inside a node, m is the number of nodes, and n refers to the number of virtual machines (VMs) running slave processes, which is obtained by $c \times m$.

  - Action1: there is a new compute node with c VMs, each one with a new application process that has an IP and a unique identification.
  - Action2: request permission to consolidate a compute node and its VMs.
- A single application process writes to the shared area, whereas the manager reads from it:
  - Action3: this gives permission to consolidate the previously requested node.

AutoElastic offers horizontal cloud elasticity using the replication technique. In the activity of enlarging the infrastructure, the manager allocates a new compute node and launches new VMs on it using an application template for slave processes. The bootstrap of a VM is ended with the execution of a slave process that will perform requests for the master. The instantiation of VMs is controlled by the manager. Only after they are running does the manager notify the master through Action1 (Figure 2). The consolidation procedure increases the efficiency of resource utilization (but not by partially using the available cores) and provides better management of energy consumption. In particular, Baliga *et al.* [37] claim that the number of VMs in a node is not an influential factor for energy consumption, but instead only whether a node is turned on. Thus, Action2 and Action3 are responsible for controlling the consolidation of a node without causing an application crash. To accomplish this, Action3 is signaled at the beginning of a loop because this point indicates a consistent global state for the distributed system.

AutoElastic's elasticity approach goes against the sentence claimed by Spinner *et al.* [38], who affirm that only vertical scaling is suitable for HPC scenarios because of inherent overhead related to the complementary approach. As presented in Figure 2, asynchronous elasticity is a crucial feature to enable horizontal and automatic resource reorganization without prohibitive costs. The master continues its execution when scaling out operations occur, being informed by the AutoElastic Manager only when the resources (including the slave processes on them) are completely up. From an application perspective, the HPC code is unaware of elasticity triggering and duration, being notified only when new resources are completely delivered. In contrast to the operations management of scaling out, underutilized resources are securely detached from the application using the sequence of Action2 and Action3, lasting at maximum a single application loop to accomplish this detaching (see Section 3.2 for details). Furthermore, we opted to use horizontal elasticity because resource resizing in a vertical approach is always limited to the nominal capacity of a single node, including primarily CPU clock and memory.

3.2. Writing the parallel application

AutoElastic acts at the middleware level, not imposing any modification in application code. However, the user must write his or her application following a set of rules, as we will discuss in this section. AutoElastic explores data parallelism on iterative message-passing applications. Currently, it works with master-slave applications, which is a parallel programming model extensively used in the following contexts: genetic algorithms, Monte Carlo technique, geometric transformations for 2D and 3D images, asymmetric cryptography and SETI@home-like applications [3]. However, we

DOI: 10.1002/cpe

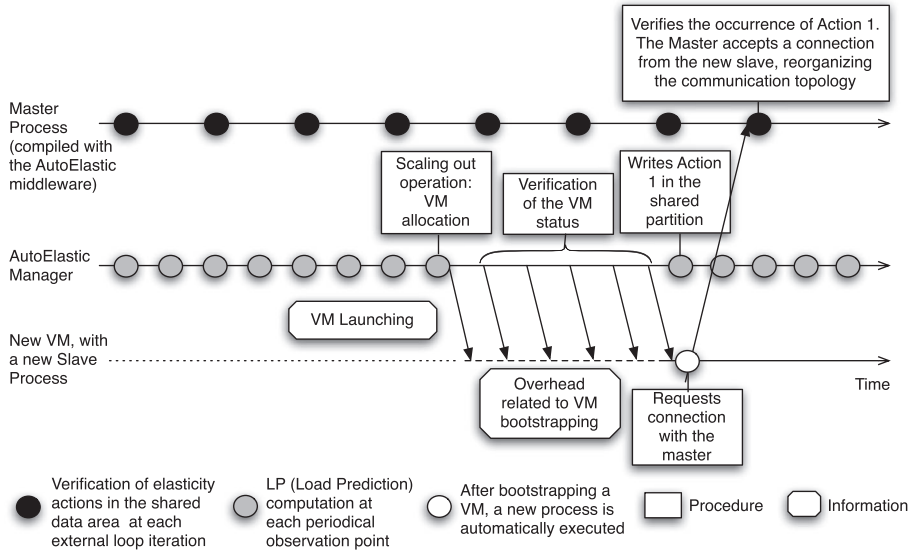


Figure 2. Functioning of the master, the new slave and the AutoElastic manager to enable asynchronous elasticity.

emphasize that the framework allows the existing processes of the HPC application to know the identifier of the new instantiated processes. This enables an all-to-all communication topology and support, for example, of bulk synchronous parallel and pipeline programming models. The AutoElastic parallel applications follow the multiple program multiple data principle, in which master and slave processes have different executable code; each is mapped to a different VM template. The intent is to offer application decoupling for processes with different purposes, enabling flexibility and facilitating the implementation of elasticity.

Figure 3(a) and (b) presents 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 by slave processes. AutoElastic works with the following MPI 2.0-like communications directives,[**] as highlighted in Figure 3: (i) publication of a connection port; (ii) looking for a server, adopting a connection port as a starting point; (iii) connection request; (iv) connection accept; (v) disconnection request; and (vi) pairwise send/receive data. Different from the approach in which a master launches processes using the so-called spawn () directive, AutoElastic acts in accordance with the MPI 2.0 approach to support dynamic process creation: socket-based point-to-point communication.

The transformation of a non-elastic application into an elastic one can be modeled at the PaaS level by one of the following three 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 lines 2 and 3; (iii) a wrapper for the function in line 3 of Figure 3 (a) can be developed for procedural languages. Independent of the strategy, the code required for elasticity is simple, as shown in Figure 3 (c). First, we must verify whether there is a new action from the AutoElastic manager in the shared data area. If Action1 has been activated, the master process reads the information concerning the new slaves and knows that it must expect new connections from them. In the case of Action2, the master removes from its group the processes that belong to a specific node. After doing that, it triggers Action3.

Although the design of AutoElastic considers master-slave applications, the iterative modeling and the use of MPI 2.0-like directives facilitates both the addition and removal of processes, and the establishment of completely new and arbitrary topologies. At the implementation level, it is possible

[**] 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.

DOI: 10.1002/cpe

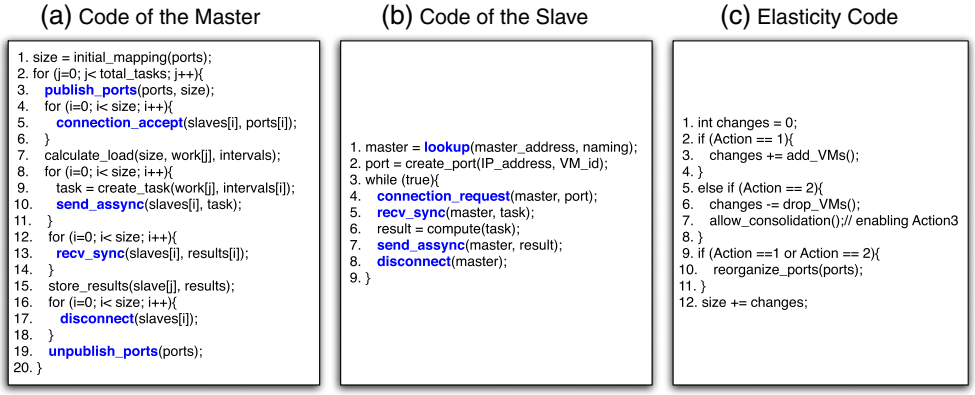


Figure 3. (a) and (b): multiple program multiple data-like parallel application model supported by AutoElastic; and (c) elasticity code to be inserted transparently by AutoElastic in the code of the Master when considering the user viewpoint.

to optimize connection and disconnection procedures if a particular slave process remains active 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.

3.3. Cloud elasticity monitoring and decision function

As in [39] and [40], the AutoElastic manager performs data monitoring periodically. Considering that HPC applications are known as being compute intensive, this version of AutoElastic focuses only on the CPU metric; specifically, on the CPU load of each VM. Hence, as a reactive-based elasticity controller, the manager obtains the CPU metric, applies the idea of moving average based on a time series of CPU loads and compares the final metric with upper and lower thresholds. More precisely, we employ a moving average [22] in accordance with Equations (1) and (2).

$$LP(i) = \frac{1}{n} \times \sum_{j=0}^{n-1} MA(i,j) \tag{1}$$

where

$$MA(i,j) = \frac{\sum_{k=i-z+1}^{i} Load(k,j)}{z} \tag{2}$$

for $i \geqslant z$.

$LP(i)$ returns a CPU load prediction when considering the execution of n slave VMs in the manager's observation number i. In this equation, j is the index of a particular VM. To accomplish the LP calculus, $MA(i,j)$ informs the CPU load of VM j at the i^{th} observation. Equation (2) uses a moving average by considering the last z observations of the CPU load $Load(k,j)$ over the VM j, where $i - z + 1 \leqslant k \leqslant i$. Finally, Action1 is triggered if LP is greater than the upper threshold, whereas Action2 is thrown when LP is less than the lower threshold. The use of a moving average is effective in avoiding VM allocation thrashing. Figure 4 shows a situation in which elasticity occurs if the load is captured as a single and immediate value. However, our approach amortizes the occurrence of sudden peaks or drops against a set of past observations, thus avoiding unnecessary resource reorganizations.

3.4. Elastic speedup and elastic efficiency models

Speedup (S) can be defined for two different types of values, throughput and latency, being commonly defined by the division of M_{old} by M_{new}. M here refers to the value of a metric, and *old* and *new* represent the execution without and with the improvement, respectively. Specifically, one of the

DOI: 10.1002/cpe

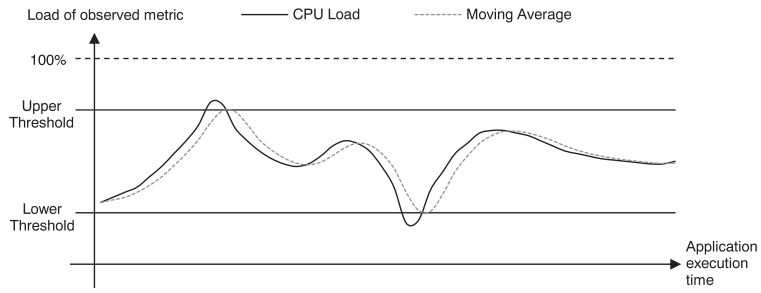


Figure 4. This scenario illustrates situations in which allocation and deallocation actions occur, but they could be neglected for thrashing avoidance when a moving average is used.

most common measurements in computer architecture – the execution time of a program – can be considered a latency quantity. Thus, Equation (3) presents the speedup S as a function of the number of processors p, such that *t(1)* and *t(p)* express the sequential and the parallel processing times. Notice that speedup is a unit-less quantity (the units cancel). Equation (4) denotes the efficiency of a parallel system, describing the fraction of the time that is being used by processors p for a given computation. Efficiency is occasionally preferred over speedup because the former represents an easy means to observe how well parallelization is working.

$$S(p) = \frac{t(1)}{t(p)} \tag{3}$$

$$E(p) = \frac{S(p)}{p} \tag{4}$$

Aiming at observing the possible gain with cloud elasticity for HPC applications, we propose an extension of the previously mentioned speedup and efficiency using these term definitions: ES and EE. Both ES and EE are explored here in accordance with horizontal elasticity, in which the instances can scale either out or in with the application demands. Furthermore, we consider a homogeneous system in which each VM runs in 100% of a CPU core, regardless of the cloud level (infrastructure as a service, PaaS, or software as a service). ES is calculated by the function *ES(n,l,u)* according to Equation (5), where n denotes the initial number of VMs and l and u represent the lower and upper bounds for the quantity of VMs, respectively. t_{ne} and t_e refer to the conclusion times of an HPC application that was executed without and with elasticity support, respectively. Here, t_{ne} is obtained with the lowest possible number of VMs, in our case l, providing an analogy with sequential execution in the standard speedup.

$$ES(n, l, u) = \frac{t_{ne}(l)}{t_e(n, l, u)} \tag{5}$$

where $l \leqslant n \leqslant u$.

Function *EE(n,l,u)* in Equation (6) computes EE. Its parameters are the same as the preceding function ES. Efficiency represents how effective the use of resources is, being the entity positioned last as denominator in the formula. Unlike Equation (4), the resources here are malleable; therefore, we designed a mechanism to reach a single value for them coherently. To accomplish this, EE assumes the execution of a monitoring system that captures the time spent on each configuration of VMs. Equation (7) indicates use of the resources, where $pt_e(i)$ is the application's partial time when running over i VMs. Logically, if elasticity actions do not occur, $pt_e(i)$ and $t_e(n, l, u)$ values will be identical for $i = n$. Equation (6) presents parameter n in the numerator, which is multiplying the ES. More precisely, this occurs because $ES(n, n, n)$ is always equal to 1 and $Resources(n, n, n)$ equal to n, so n in the numerator returns an EE of 100%.

$$EE(n, l, u) = \frac{ES(n, l, u) \times n}{Resource(n, l, u)} \tag{6}$$

DOI: 10.1002/cpe

where

$$Resource(n, l, u) = \sum_{i=l}^{u} \left(i \times \frac{pt_e(i)}{t_e(n, l, u)} \right) \quad (7)$$

The denominator in Equation (6) was added to capture each infrastructure configuration (from l to u VMs) and its participation in the entire execution, presenting the sum of the partial values as the final value for $Resource(n, l, u)$. The use of a flexible number of resources helps to provide better parallel efficiency when comparing elastic and non-elastic parallel executions. At a glance, there are two approaches to make viable the aforementioned statement: (i) the elastic execution presents an equal, or even a bit greater, execution time when compared with a non-elastic configuration, but we employ a smaller set of resources to reach the first case. (ii) Even devoting a larger number of resources because the CPU-bound HPC application demands them, the gain in the time perspective outperforms the investment in resources. Although scaling in is a key operation to accomplish (i), scaling out is essential for (ii).

3.5. Energy consumption models

The idea of obtaining better application performance only with an elastic-assisted execution occasionally is not sufficient for users and cloud administrators. For example, an elasticity manager that provides resource over-provisioning with bad resource consumption and bad accuracy between VM allocation and application demand could cause problems for both users and administrators. Users would pay for a greater number of resources, not effectively used, in accordance with the pay-as-you-go paradigm. Administrators can suffer from resource sharing among users, besides wasting energy. Specifically, the power consumption of data centers doubled from 2000 to 2005 worldwide, going from 70.8 billion kWh to 152.5 billion kWh [41]. EE (Section 3.4) already indicates resource consumption, but we cannot precisely draw conclusions about energy data. In this context, we have modeled two power models to extract data about energy consumption, exploring the relationships among energy consumption, resource consumption and performance thereafter. In the succeeding discussions, both energy models are described in detail.

3.5.1. Energy model A – energy forecasting based on real traces. Deploying energy sensors or wattmeters can be costly if not undertaken at the time the entire infrastructure (i.e., cluster or data center) is built, besides being time consuming as the infrastructure scales up. An alternative and less expensive solution is to use energy models to estimate the consumption of components or of an entire infrastructure [42]. Good models should be lightweight and should not interfere with the energy consumption that they try to estimate. Thus, energy model A explores energy data captured in a small set of compute nodes in the cloud, thus formulating an equation to extend the results to an arbitrary set of homogeneous nodes. Specifically, we follow the methodology used by Luo *et al.* [13], which consists of three phases:

(i) Collect samples of resource usage and machine energy consumption using a smart power meter. In our case, we used a Minipa power meter model ET-4090 and collected data from more than 8000 samples using a composite load that may consume multiple types of cloud resources to represent real cloud applications [43].
(ii) Perform regression methods to generate the energy model to be used later.
(iii) Test the model on a different set of data collected from other homogeneous machines to validate whether the model works properly for the machines.

First, we collected samples of CPU, main memory, disk, network, and cache memory to determine which factors have higher weight in server energy consumption. The goal was to minimize the amount of resources to be collected, consequently reducing the load on the served imposed by the data collector. To do so, we analyzed the data collected using the Kaiser criterion [44], which determines that any component with an eigenvalue less than 1 can be discarded. The analysis resulted in a script that collects only CPU and memory usage. Our result is in line with previous studies [42,45] that present the CPU as the main villain for energy consumption inside a compute node.

DOI: 10.1002/cpe

Using CPU, main memory, and instantaneous power consumption data, we applied principal component regression to more than 8000 samples obtained from a single node. After that, we predicted the same amount of power samples using CPU and memory data collected from another node with the same hardware configuration. Comparing these power samples with the samples collected with the power meter as seen in Figure 5, we obtained a mean and median accuracy of 97.15% and 97.72%, respectively.

After the application execution, we input the CPU and main memory values into the trained model to obtain instantaneous power consumption measured in Watts (W). The instantaneous power consumption values are integrated to obtain what is called energy-to-solution [46] represented in Joules (J). This value defines how much energy has been spent during the application execution.

3.5.2. Energy model B – empirically analyzing energy through the resource usage. To improve the reproducibility of the results and provide an easier way to measure the energy consumed during application execution, this paper also presents an empirical energy model. This model relies on the close relationship between energy and resource consumption as presented by Orgerie *et al.* [42]. In this model, we use Equation (8) to create an index of the resource usage, with l and u with the same meanings described in Section 3.4. Here, we use the same ideas of the pricing model employed by Amazon and Microsoft; they consider the number of VMs at each unit of time, which is normally set to an hour. $pt_e(i)$ presents the time that the application executed with i VMs. Therefore, our unit of time depends on the measure of pt_e (in minutes, seconds, or milliseconds, and so on) in which the final intent is to sum the number of VMs used at each unit of time. For example, considering a unit of time in minutes and an application completion time of 7 min, we could have the following histogram: 1 min (2 VMs), 1 min (2 VMs), 1 min (4 VMs), 1 min (4 VMs), 1 min (2 VMs), 1 min (2 VMs), and 1 min (2 VMs). Finally, 2 VMs were used in 5 min (partial resource equal to 10) and 4 VMs in 2 min (partial resource equal to 8), summing to 18 for Equation (8). Thus, Equation (8) analyzes the use from l to u VMs, considering the partial execution time on each infrastructure size. To the best of our understanding, the energy index here is relevant to two situations: (i) for comparison among different elastic-enabled executions by varying l or u and (ii) to analyze the accuracy between the curves of energy models A and B.

$$Resource'(l,u) = \sum_{i=l}^{u} (i \times pt_e(i)) \tag{8}$$

3.6. Cost model

We use a cost model to estimate elasticity viability by multiplying the application time by a particular energy model, A or B (Equation (9)). Our notion of cost in Equation (9) represents an adaptation of the cost of a parallel computation [47], now to support elastic environments. We only change the number of processors by the value obtained from one of the energy models. The final idea consists of obtaining a better cost when enabling the AutoElastic's elasticity feature in comparison with an

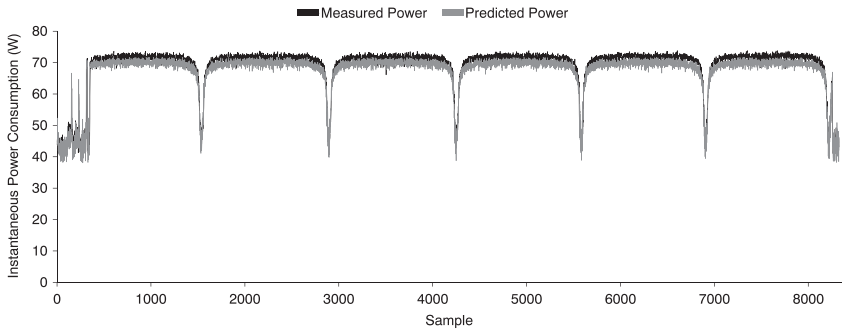


Figure 5. Accuracy of instantaneous power consumption samples predicted with energy model A compared with measured data.

DOI: 10.1002/cpe

execution of a fixed number of processes. In other words, a configuration solution may be classified as bad if it is capable of reducing the execution time by one-half with elasticity but spends four times more energy, thus increasing costs. Therefore, considering the values of the cost function in elastic and non-elastic environments, the objective is to preserve the truth of inequality(10).

$$Cost = App\_Time \times Energy\_Model \tag{9}$$

then, the goal is to obtain

$$Cost_{\alpha} \leqslant Cost_{\beta} \tag{10}$$

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

4. EVALUATION METHODOLOGY

The evaluation methodology consists of developing an HPC scientific application, then testing it with and without AutoElastic. We divided this section into three subparts, exploring application details in Sections 4.1 and 4.2 and prototype and execution details in Section 4.3.

4.1. HPC application modeling

The application used in the tests computes the numerical integration of a function *f(x)* in a closed interval $[a, b]$. In general, numerical integration has been used in two ways: (i) as benchmark for HPC systems, including multicore, Graphical Processing Unit (GPU), cluster and grid architectures [48,49] and (ii) as a computational method employed on simulations of dynamic and electromechanical systems [50,51]. The first case explains why we are using numerical integration to evaluate the AutoElastic model. Here, we are using the composite trapezoidal rule from a Newton–Cotes postulation [50] to implement the application. The Newton–Cotes formula can be useful if the value of the integrand is given at equally spaced points. Consider the partition of interval $[a, b]$ into s equally spaced subintervals, each one with length h ($[x_i; x_{i+1}]$, for $i = 0, 1, 2, \ldots, s-1$). 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 (11). Equation (12) shows the development of the integral in accordance with the Newton–Cotes postulation. This equation is used to develop parallel application modeling.

$$\int_a^b f(x)\,dx \approx A_0 + A_1 + A_2 + A_3 + \ldots + A_{s-1} \tag{11}$$

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

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + f(x_s) + 2.\sum_{i=1}^{s-1} f(x_i)\right] \tag{12}$$

The values of x_0 and x_s in Equation (12) are equal to a and b, respectively. In this context, s equals 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 do 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.

4.2. Defining the application's workload through different load patterns

Aiming at analyzing the parallel application on different input loads, four patterns were developed and named constant, ascending, descending, and wave. The idea of using different patterns, or workloads, for the same HPC application is widely explored in the literature to observe how the input load can impact in points of saturation, bottlenecks, and resource allocation and deallocation [52–54].

DOI: 10.1002/cpe

Table I. Functions to express different load patterns.

Load	Load function	Parameters			
		v	w	t	z
Constant	$load(x) = \frac{w}{2}$	—	1,000,000	—	—
Ascending	$load(x) = x * t * z$	—	—	0.2	500
Descending	$load(x) = w - (x * t * z)$	—	1,000,000	0.2	500
Wave	$load(x) = v * z * sen(t * x) + v * z + w$	1	500	0.00125	500,000

In *load(x)*, x is the iteration index at application runtime.

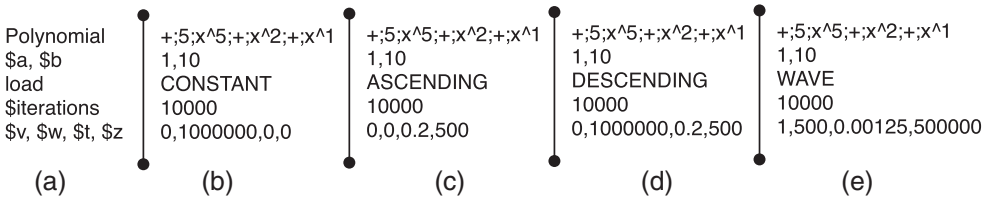


Figure 6. (a) Template of the input file for the tests; (b)–(e) instances of the template when observing the load functions in Table I.

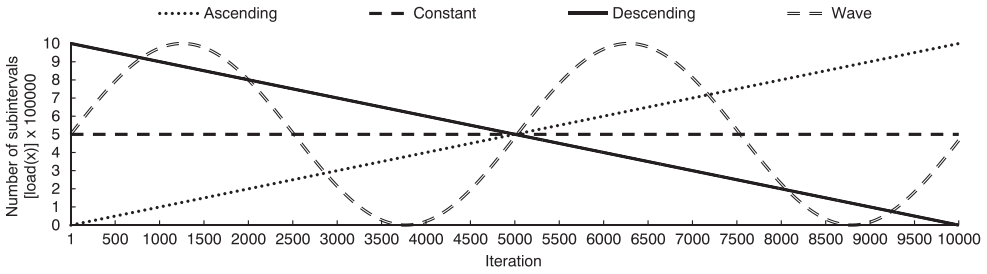


Figure 7. Graphical vision of the load patterns.

Table I and Figure 6 show the equation of each pattern and the template used in the tests. The iterations in this figure represent 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 place attention on the load variations and not on the result of the numerical integration itself.

Figure 7 shows a graphical representation of each pattern. The x-axis in the graph of Figure 7 expresses the functions (each iteration represents a function) that are being tested, whereas the y axis informs the respective load. Again, the load represents the number of subintervals s between limits a and b, which in this experiment are 1 and 10, respectively. A greater number of intervals is associated with a greater computational load for generating the numerical integration of the function. For simplicity, the same function is employed in the tests, but the number of subintervals for the integration varies.

4.3. Prototype implementation, execution parameters, and evaluated metrics

We implemented an AutoElastic prototype for private clouds using OpenNebula v4.2. We implemented the AutoElastic manager and the parallel application in Java. Two image templates for the VMs 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.

DOI: 10.1002/cpe

There are some technical decisions that are imposed in the implementation of the AutoElastic prototype: (i) utilization of NFS 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, 70% and 90% were used for the upper threshold, whereas 30% and 50% were used for the lower threshold. Examining decision (i), the manager uses the binary Secure Copy (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.

Our experiments were conducted using the OpenNebula private cloud with six (one front-end and five nodes) homogeneous nodes. We used 2.9 GHz dual-core nodes with 4 GB of RAM and an interconnection network of 100 Mbps. Each load was executed when considering AutoElastic with and without enabling the elasticity feature. In the case in which elasticity is enabled, the thresholds used were 70% and 90% for the upper threshold and 30% and 50% for the lower threshold. Based on a simple combination, all loads were tested four times when elasticity was enabled, where four is the number of combinations of upper-selected and lower-selected thresholds. All executions started from the same scenario, which consisted of a single node with two VMs (that is the same as the number of cores in the node). In the elastic executions, the cloud can scale out to a limit of five nodes (10 VMs) defined by an SLA. Concerning the metrics, we capture the application execution time, data related to energy models A and B, and the costs considering each of the energy models, ES and EE. Moreover, we present a profile analysis on resource and energy consumption, and performance graphs in which we highlight particular behavior in the light of energy and application modeling.

5. RESULTS

We divided this section into three moments:

(i) a quantitative analysis in Section 5.1, including all parameters and metrics, which aims at verifying possible energy and performance arrangements;
(ii) Section 5.2 presents graphs highlighting the relationship between performance and energy consumption; and
(iii) an overall discussion appears in Section 5.3 that emphasizes the main achievements and drawbacks, in terms of both evaluation methodology and functioning, of the AutoElastic prototype.

5.1. Analyzing elastic speedup, elastic efficiency, and cost

Table II presents the evaluated measures considering different threshold settings and load patterns. Table III acts a complementary data source, in which we can observe resource consumption in the light of the combination of thresholds. Considering the ascending, constant and wave (this last presents two phases of load elevation) load patterns, an upper threshold equal to 70% was primarily responsible for proving better results in terms of execution performance. This threshold implies a more reactive system because the average CPU load of the system will not exceed this limit. Considering that the tested application is CPU intensive, VM allocations happen more quickly with this threshold. This procedure triggers load balancing among a larger number of slave processes and, consequently, owing to the computation grain that remains viable, implies a reduction in the application time. Considering the aforementioned load patterns, the upper threshold of 70% was engaged on achieving not only the best application times but also the values of the ES index. Conversely, this combination of this threshold over the ascending, constant, and wave patterns follows the traditional statement of parallel computing, that is, there is a curve along which a greater number of resources entails a better application time, but efficiency decreases with time. In other words, this context of load pattern and threshold resulted in a lower EE and in a greater energy consumption as revealed

DOI: 10.1002/cpe

Table II. Results of all scenarios, including elasticity support, load patterns and thresholds.

				Results						
		Threshold		Time	Energy model	Energy model*	Energy model A-based cost:	Energy model B-based cost:		
Mode	Load	Upper	Lower	(s)	A (KJ)	B	"Cost A"	"Cost B"	ES	EE
AutoElastic with elasticity support	Ascending	70	30	1869	395.88	13428	739899.42	24760512	2.280	0.6433
			50	**1858**	395.96	13284	**735695.38**	**24681672**	**2.293**	0.6415
		90	30	2965	*341.28*	*10404*	1011886.38	30847860	1.437	*0.8191*
			50	3088	341.93	10418	1055894.77	32170784	1.380	0.8180
	Constant	70	30	**1883**	399.28	13422	**751844.97**	**25273626**	**2.271**	0.6373
			50	1914	399.21	13440	764088.74	25724160	2.235	0.6365
		90	30	2730	*348.53*	*10794*	951476.42	29489460	1.567	*0.7925*
			50	2737	348.79	10802	954631.29	29543178	1.563	0.7919
	Descending	70	30	1929	451.49	15930	870921.55	30728970	2.222	0.5381
			50	2787	353.75	11032	985908.99	30746184	1.538	0.7770
		90	30	**1926**	434.64	15132	**837122.62**	**29144232**	**2.225**	0.5665
			50	2761	*349.19*	*10844*	964100.40	29940284	1.552	*0.7905*
	Wave	70	30	**1959**	451.17	15888	**883848.92**	**31124592**	**2.193**	0.5408
			50	2053	453.26	15914	930552.55	32671442	2.093	0.5399
		90	30	3050	362.04	11312	1104229.61	34501600	1.409	0.7595
			50	3037	*359.48*	*11188*	1091748.17	33977956	1.415	*0.7680*
AutoElastic without elasticity support	Ascending			4261	289.43	8522	1233242.21	36312242	1	1
	Constant			4277	291.37	8554	1246175.30	36585458	1	1
	Descending			4286	290.86	8572	1246604.79	36739592	1	1
	Wave			4296	291.30	8592	1251415.43	36911232	1	1

Here, we highlight two arrangements: the values in bold and italic represent the best results for each load pattern when elasticity was enabled. Values in bold represent the best cases for performance (time and elastic speedup (ES)), whereas values in italic prioritize efficiency (energy models A and B, and elastic efficiency (EE)).
* According to Equation 8, the unit here is VMs × Time (in seconds).

Table III. Resource consumption (given as the number of used VMs) with elastic and non-elastic executions in accordance with Equation (7).

	Thresholds		Load patterns			
Mode	Upper	Lower	Ascending	Constant	Descending	Wave
AutoElastic with elasticity support	70	30	7.088	7.128	8.258	8.110
		50	7.150	7.022	3.958	7.752
	90	30	3.509	3.944	7.857	3.709
		50	3.374	3.957	3.928	3.684
AutoElastic without elasticity support	—	—	2	2	2	2

Each virtual machine (VM) is mapped to a particular core of a computational node; therefore, the values here also indicate CPU consumption of the executions.

in both energy models A and B. Table III corroborates this sentence, in which a threshold equal to 70% was responsible for a greater resource consumption of the evaluated load patterns.

In contrast to the elastic execution over the ascending, constant, and wave functions, in which the performance was dictated by the upper threshold of 70%, the behavior of the descending pattern is highly influenced by the lower thresholds. Specifically, the value of 30% for the lower threshold was responsible for maintaining the allocated resources for longer when compared with 50%; thus, 30% led to a better application time and ES indexes. Exploring the same idea discussed earlier, the computation grain is sufficiently large to direct us to obtain better performance results when a large number of results is considered. Conversely, EE provides the worst results when testing the application with the descending function with the threshold equal to 30%. Logically, this happens because the resources are, on average, sparingly used, that is, they no longer approach close to 100% of utilization. Thus, the employment of 50% for the lower threshold allows reaching better values for

DOI: 10.1002/cpe

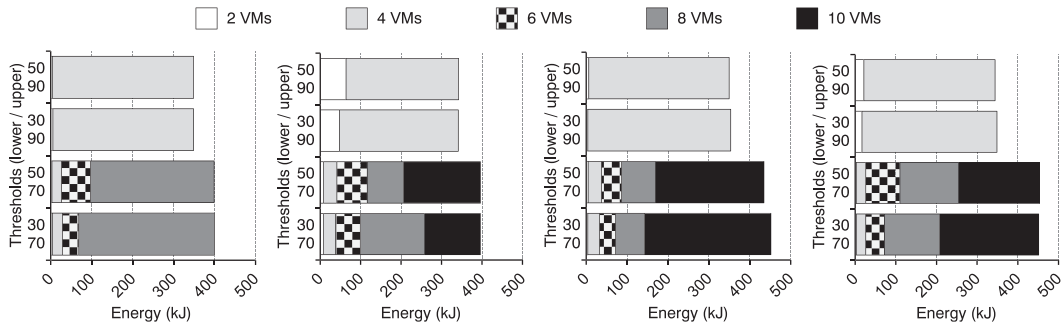
EE because the resources are deallocated sooner. This statement is presented in the data of Table III, in which resource consumptions of close to 4 and 8 are achieved with the thresholds 50% and 30%, respectively.

Regardless of the threshold used, the AutoElastic with elasticity support was responsible for providing better values for costs A and B in comparison with the non-elastic execution. Here, Cost A and cost B refer to Equation (9) when using energy models A and B, respectively. Considering that our notion of cost concerns a multiplication of energy model and application time and that we obtained better performance results, we can affirm that AutoElastic addresses elasticity in a non-prohibitive way. In other words, AutoElastic's algorithms are responsible for proper management of the number of resources and the respective time used by each of them.

5.2. Performance and energy comparison through graphs and profiles

Figure 8 presents an application energy consumption profile in accordance with energy model A. We observed that greater application energy consumption implied a greater number of employed VMs. Starting from a single node (2 VMs), the application with ascending, descending, and wave functions allocated up to five nodes (10 VMs), whereas constant stops at four nodes. The variation of the upper threshold directly affects total consumed energy. The value of 70% implies the triggering of VM allocations sooner because the system load will never exceed this limit. Explaining the same situation from a different perspective, we have that a greater upper threshold results in better energy consumption. The lower threshold does not significantly affect the execution; we can only observe that the value of 50% entails a slightly better distribution of VM usage among the load functions. Figure 9 illustrates an application time profile in accordance with the number of allocated VMs. As shown in Table II, performance is inversely proportional to energy consumption. In this context, an upper threshold equal to 90% does not offer application reactivity, as seen in the eight bars that represent this value (in Figure 9, we have four bars for the 50×90 combination and four other bars for the 30×90 setting). Even working with a CPU-hungry application, the load is unlikely to exceed this value. Consequently, resource reorganization is postponed and may never occur. Finally, Figure 10 depicts five graphs comparing total energy consumption (model A) and total elapsed time in both non-elastic and elastic executions. This figure demonstrates again the inverse proportion between these metrics.

Figure 11(a) presents an execution graph highlighting peaks and sudden drops of power consumption when analyzing the descending function for the four combinations of thresholds. In part (i), we have a host allocation (and 2 VMs inside it), which results in a sudden drop followed by energy consumption increasing thereafter. In addition, we can observe oscillations during the bootstrapping of the VMs, as seen in part (ii). After this procedure, the manager writes Action1 in the shared data area, and the master process knows that there are new resources, and consequently processes, to be used. Part (iii) of Figure 11(a) explores this situation, in which the master reorganizes the communi-



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

Figure 8. Using energy model A to capture energy profiles of different load patterns when varying the thresholds. The number of employed virtual machines (VMs) is gradual, 'always' with an increment of 2 VMs at each scaling out operation.

DOI: 10.1002/cpe

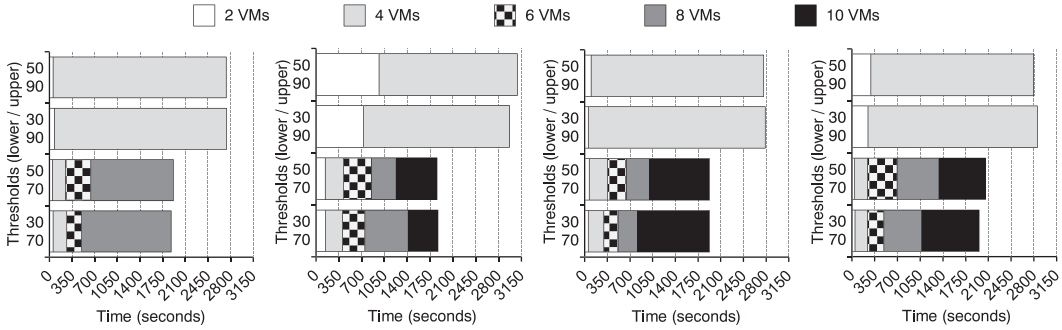


Figure 9. Performance profiles and resource utilization when considering different load patterns and thresholds. The number of employed virtual machines (VMs) is gradual, always with an increment of 2 VMs at each scaling out operation.

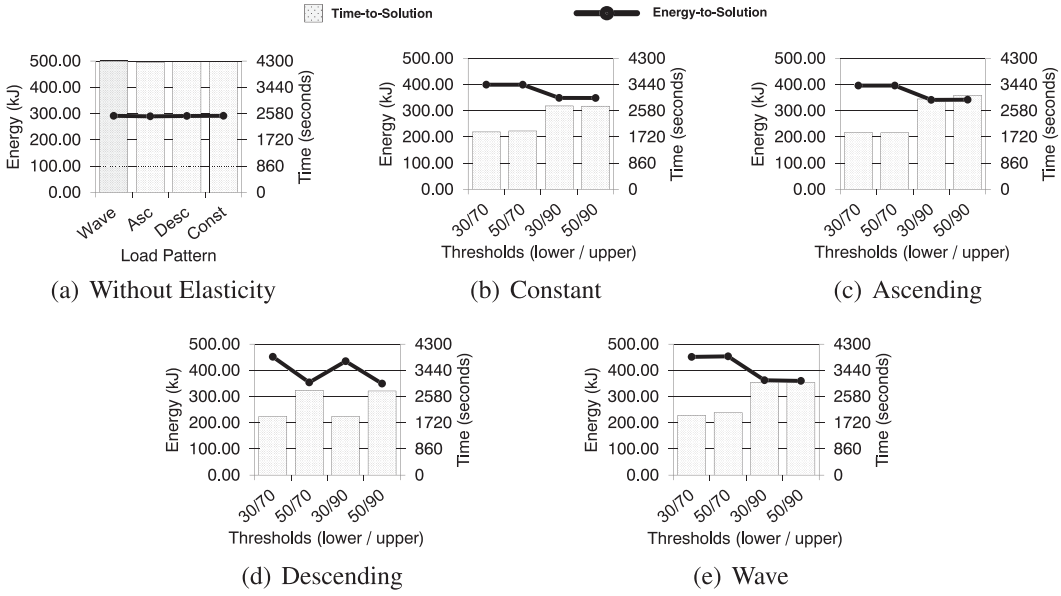


Figure 10. Joint analysis of energy and execution time: (a) Load patterns without elasticity; (b–e) constant, ascending, descending, and wave load patterns when enabling elasticity and varying the thresholds.

cation topology, causing an abrupt decline in power consumption because no CPU usage occurs. The application proceeds in a decreasing curve of load; therefore, resources are deallocated as illustrated in part (iv). Here, we have the shutdown of the VMs, and the node is subsequently consolidated. Figure 11 explores the power consumption of model A for the constant, ascending, descending, and wave load functions. The threshold of 70% is decisive in determining energy consumption because it enables more reactiveness to the possibility of allocating more resources. As discussed earlier, conversely, this same threshold is also responsible for the best performance rates.

Figure 12 illustrates the power consumption in Watts in accordance with model A when elasticity actions are disabled. In this context, a single node with two VMs is being used to host slave processes. Here, we observe that the simple fact of turning on the compute node (running the Ubuntu Linux Operating System and AutoElastic software) spends approximately 40 W. Any computation activity causes an elevation of this index to the interval (40–71). Although the ascending function grows slowly, the power consumption here increases quickly up to the upper bound of the interval. The same behavior appears in the descending and wave functions.

DOI: 10.1002/cpe

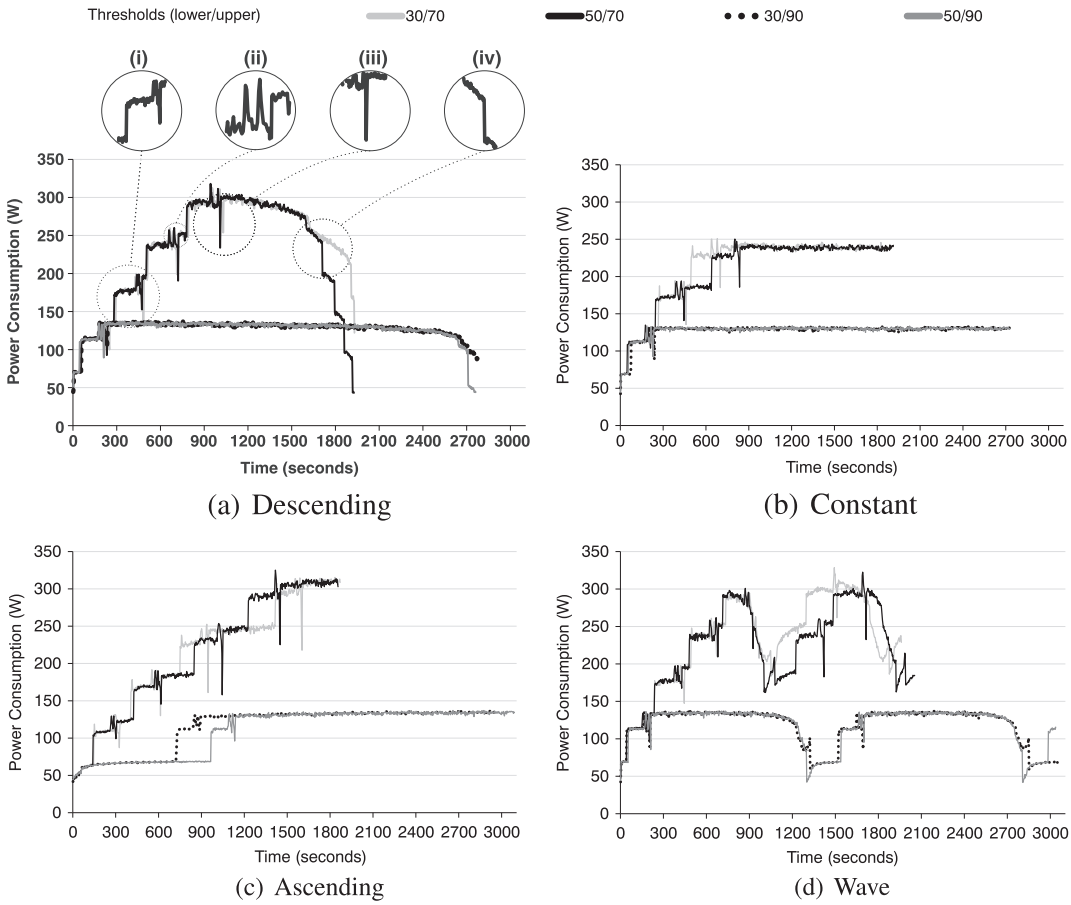


Figure 11. Power consumption behavior of different load patterns with varying thresholds. Specifically, in (a) we highlight the following moments: (i) host allocation, (ii) virtual machines booting, (iii) processing stop to incorporate new resources, and (iv) host deallocation.

5.3. Discussion

AutoElastic uses the metric CPU load to drive elastic actions; thus, the upper and lower thresholds are decisive in obtaining performance and resource allocation for HPC applications. As presented in Figures 9–11, resource allocation and the application behavior are also important to energy consumption; greater CPU usage implies greater energy consumption. Moreover, an analysis of the aforementioned figures and Tables II and III indicates the strong relationship between energy models A and B. In other words, the use of malleable resources as measured in model B is proportional to the power consumption in Joules and Watts, as returned in model A. Concerning the effect of the thresholds on performance and energy, here we will concentrate our analysis on two complementary load patterns – ascending and descending – as follows:

- Ascending load pattern: An upper threshold far from 100% is decisive for achieving performance and better values for ES because the application will never execute in an overloaded state. Conversely, an upper threshold close to 100% (in our case, 90% is the largest value) delays the resource allocation, being more effective in using already allocated resources, thus enlarging EE. Here, the lower threshold does not have a decisive effect on EE and ES because the application's demand is always increasing over time.
- Descending load pattern: Resource deallocation is delayed when using a lower threshold equal to 30%, favoring ES because the CPU-intensive application will run with a greater number of resources. The use of 50% for the lower threshold is useful to deallocate resources sooner than

DOI: 10.1002/cpe

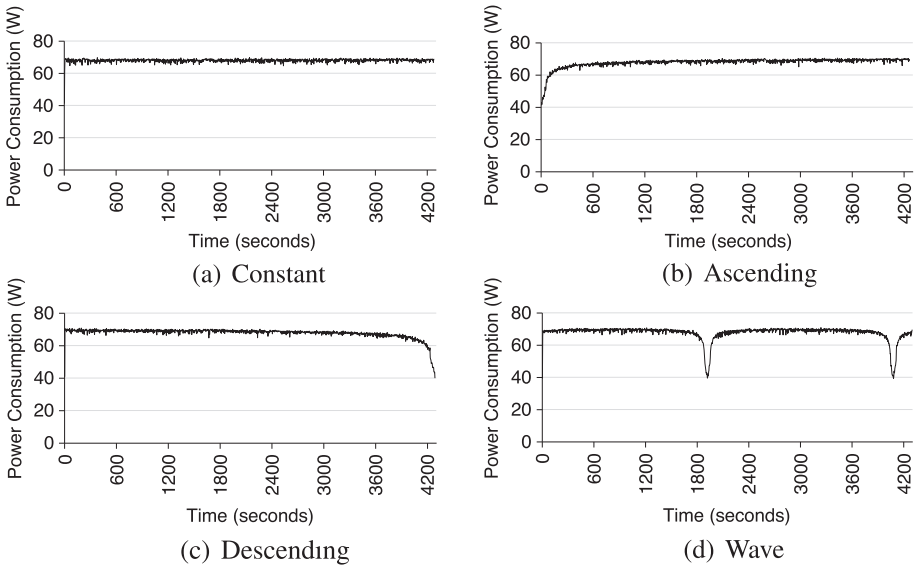


Figure 12. Power consumption behavior of different load patterns without elasticity.

when employing 30%, as soon as the manager detects a system load lower than 50%. However, slave processes that remain active are responsible for using CPU resources closer to 100% because the computational grain (ratio between computation and communication activities) increases when reducing the number of processes.

With the joint analysis of the two load patterns and an infrastructure of up to 10 CPUs, we conclude that using more CPUs is better from a performance perspective, although their capacities may be underused. This sentence may not be correct for larger infrastructures because a greater number of processes implies a greater overhead from communication actions. The setup of the thresholds to drive resource allocation is a user procedure. However, he often does not know the application behavior, or it may not be easily predictable in advance. Despite its flexibility, this situation can be considered an AutoElastic point of weakness. Thus, an alternative is to work with elastic thresholds; a set of initial values is adapted at runtime based on application traces. This feature will be explored as future work.

We use horizontal elasticity to support infrastructure reorganization. To accomplish this in a non-prohibitive way from the application viewpoint, asynchronous elasticity is responsible for not blocking the processes while new resources are being started. However, we envision the problem of reactiveness as follows: AutoElastic manager detects the need to allocate more resources at time t, and actually performs this thereafter, but delivers them to the application at time $t + z$. However, at instant $t + z$, the resources may not be more useful to the current state of the application. The use of the moving average concept was helpful to avoid obtaining this situation in our tests, but we expect to see the problem when highly irregular applications are tested. Although varying the workload, all load patterns employed in the evaluation were based on polynomial functions, not being classified as irregular. Finally, horizontal elasticity was a design decision when developing AutoElastic. As said earlier, asynchronous elasticity makes it viable for HPC applications. Furthermore, in contrast to resizing VMs with vertical elasticity, the horizontal approach is useful for enabling resource deallocation and for enlarging the infrastructure beyond the limits of a single resource.

Finally, besides presenting the main scientific achievements, it is equally important to discuss the limitations of both AutoElastic model and its evaluation. In this way, we can highlight six points: (i) either the programmer or the cloud administrator must select the lower and upper thresholds at middleware level, and this procedure could be not trivial for non-cloud experts; (ii) we have used different load patterns, which represent regular workloads modeled by math equations, that is, we do not have sudden peaks when considering two consecutive iterations; (iii) the energy model A is particularly suitable for CPU-bound master-slave iterative applications like the proposed implemen-

DOI: 10.1002/cpe

tation of the numerical integration problem; (iv) as the standard definitions of speedup and efficiency, both EE and ES work fine only when considering homogeneous resources, that is, all VMs must present exactly the same hardware configuration; (v) the proposed energy model for elastic applications, named here as energy model A, has as one of its assumptions the beforehand execution of the application in at least two nodes and the power consumption capturing, so enabling us to develop the regression procedures afterward; so this task is more feasible on private clouds where we normally have access to the physical resources; and (vi) the AutoElastic manager monitors the VMs periodically, and this interval must also be set up at middleware level. Particularly, the selection of an adequate monitoring interval for the sixth topic is not an easy task because of (i) a short interval could imply on larger monitoring overheads, which can impact the functioning of the HPC application, and (ii) a large interval could imply on the loss of reactivity, delaying the detection of scaling in and out operations.

6. CONCLUSIONS

This article focused on performance and energy consumption in elastic cloud computing systems using the AutoElastic model as a substrate to discuss novel ideas. The scientific contributions of this article are threefold: (i) ES and EE; (ii) analysis of energy and performance, together with resource utilization, on elastic HPC environments; and (iii) an energy forecasting model based on real traces, here called model A. Considering the initial number of VMs and the lower and upper bounds, the article's contribution explores the traditional speedup and efficiency for parallel systems, now considered in elastic infrastructures. Although ES provides possible gains with on-the-fly resource reorganization, EE indicates the effectiveness of resource use. In other words, it would not be interesting to reach a significant ES rate, but pay an EE close to 0. One can use ES and EE to analyze algorithms either theoretically, using asymptotic runtime complexity, or in practice, to measure the effectiveness of the use of new available resources. In addition to performance and resource usage, we observe that for both financial (possible money savings for cloud administrators) and environmental reasons (reduction of carbon dioxide emissions), energy consumption has become a critical concern in designing modern cloud-based systems. Thus, our third contribution investigates the energy consumption of HPC applications using the cloud. Through real instrumentations, we provide a model to estimate energy consumption in Watts across an arbitrary set of compute nodes.

We conducted experimental evaluations by varying both lower and upper thresholds, and the load functions that dictate the workload's characteristics. Here, we highlight the main technical contributions extracted from the results: (i) we used 70% and 90% as values for the upper threshold. Seventy percent enhanced AutoElastic's reactivity, thus yielding better application times and ES indexes, but paying a larger EE. (ii) We used 30% and 50% as values for the lower threshold. Particularly for the descending function, the value of 50% brings resource deallocation sooner than 30%, thus providing us better values for EE. (iii) Our cost function, which multiplies the application time by the measure of an energy model, shows better results in favor of using elasticity when compared with static deployments, that is, AutoElastic offers this feature in a non-prohibitive way. (iv) The results emphasized the strong relationship between energy consumption (in Watts) and resource usage (computed using the number of used VMs and the time spent on each instance deployment).

Even employing the idea of a moving average to smooth sudden load peaks, AutoElastic follows the reactive elasticity principles always to answer with resource reorganization after exceeding thresholds. Therefore, future work includes a hybrid proactive and reactive solution for the next version of AutoElastic, joining ideas from reinforcement learning, neural networks, and/or time-series analysis. Moreover, we plan to develop algorithms to provide the idea of 'elastic thresholds', in which the lower and upper bounds are adaptable in accordance with historical data concerning application performance. Finally, we intend to explore further the EE concept of using fixed thresholds and the previously mentioned idea of elastic thresholds.

ACKNOWLEDGEMENT

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

DOI: 10.1002/cpe

REFERENCES

1. Lorido-Botran T, Miguel-Alonso J, Lozano J. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing* 2014; **12**(4):559–592.
2. Weber A, Herbst NR, Groenda H, Kounev S. Towards a resource elasticity benchmark for cloud environments. *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, ACM, Dublin, Ireland, 2014.
3. Raveendran A, Bicer T, Agrawal G. A framework for elastic execution of existing MPI programs. In *Proceedings of the 2011 IEEE Int. Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW '11)*, IEEE Computer Society: Washington, DC, USA, 2011; 940–947.
4. Jamshidi P, Ahmad A, Pahl C. Autonomic resource provisioning for cloud-based software. *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014*. ACM: New York, NY, USA, 2014; 95–104.
5. Petrides P, Nicolaides G, Trancoso P. HPC performance domains on multi-core processors with virtualization. *Proceedings of the 25th International Conference on Architecture of Computing Systems, ARCS'12*. Springer-Verlag: Berlin, Heidelberg, 2012; 123–134.
6. Roloff E, Diener M, Carissimi A, Navaux P. High performance computing in the cloud: deployment, performance and cost efficiency. *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, Taipei, Taiwan, 2012; 371–378.
7. Sharma U, Shenoy P, Sahu S, Shaikh A. A cost-aware elasticity provisioning system for the cloud. *Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS '11)*, IEEE Computer Society: Washington DC, USA, 2011; 559–570.
8. Guo Y, Ghanem M, Han R. Does the cloud need new algorithms? An introduction to elastic algorithms. *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, Taipei, Taiwan, 2012; 66–73.
9. Galante G, Bona LCEd. A survey on cloud computing elasticity. *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC '12)*, IEEE Computer Society: Washington DC, USA, 2012; 263–270.
10. Beloglazov A, Abawajy J, Buyya R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems* May 2012; **28**(5):755–768.
11. Tian Y, Lin C, Li K. Managing performance and power consumption tradeoff for multiple heterogeneous servers in cloud computing. *Cluster Computing* 2014; **17**(3):943–955.
12. Paya A, Marinescu DC. Energy-aware application scaling on a cloud. *CoRR*. 2013; **abs/1307.3306**.
13. Luo L, Wu W, Tsai W, Di D, Zhang F. Simulation of power consumption of cloud data centers. *Simulation Modelling Practice and Theory* 2013; **39**(0):152–171.
14. Beloglazov A, Buyya R. Energy efficient resource management in virtualized cloud data centers. *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID '10)*, IEEE Computer Society: Washington, DC, USA, 2010; 826–831.
15. Wei H, Zhou S, Yang T, Zhang R, Wang Q. Elastic resource management for heterogeneous applications on PaaS. *Proceedings of the 5th Asia-Pacific Symposium on Internetware (Internetware '13)*, ACM: New York, NY, USA, 2013; 7:1–7:7.
16. Leite AF, Raiol T, Tadonki C, Walter MEMT, Eisenbeis C, de Melo ACMaA. Excalibur: an autonomic cloud architecture for executing parallel applications. *Proceedings of the Fourth International Workshop on Cloud Data and Platforms, CloudDP '14*. ACM: New York, NY, USA, 2014; 2:1–2:6.
17. Al-Shishtawy A, Vlassov V. Elastman: elasticity manager for elastic key-value stores in the cloud. *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC '13)*. ACM: New York, NY, USA, 2013; 7:1–7:10.
18. Aniello L, Bonomi S, Lombardi F, Zelli A, Baldoni R. An architecture for automatic scaling of replicated services. In *Networked Systems*, Noubir G, Raynal M (eds). Lecture Notes in Computer Science, Springer International Publishing: Switzerland, 2014;122–137.
19. Gutierrez-Garcia JO, Sim KM. A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. *Future Generation Computer System* 2013; **29**(7):1682–1699.
20. Tesfatsion S, Wadbro E, Tordsson J. A combined frequency scaling and application elasticity approach for energy-efficient cloud computing. *Sustainable Computing: Informatics and Systems* 2014; **4**(4):205–214.
21. Martin P, Brown A, Powley W, Vazquez-Poletti JL. Autonomic management of elastic services in the cloud. *Proceedings of the 2011 IEEE Symposium on Computers and Communications (ISCC '11)*, IEEE Computer Society: Washington, DC, USA, 2011; 135–140.
22. Coutinho E, de Carvalho Sousa F, Rego P, Gomes D, de Souza J. Elasticity in cloud computing: a survey. *Annals of Telecommunications - Annales des Tlcommunications* 2015; **70**(7):289–309.
23. Islam S, Lee K, Fekete A, Liu A. How a consumer can measure elasticity for cloud platforms. *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*, ACM: New York, NY, USA, 2012; 85–96.
24. Tembey P, Gavrilovska A, Schwan K. Merlin: application- and platform-aware resource allocation in consolidated server systems. *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, ACM: New York, NY, USA, 2014; 14:1–14:14.

DOI: 10.1002/cpe

25. Wang H, Jing Q, Chen R, He B, Qian Z, Zhou L. Distributed systems meet economics: pricing in the cloud. *HotCloud '10* USENIX: Berkeley, CA 2010.
26. Garg SK, Yeo CS, Anandasivam A, Buyya R. Environment-conscious scheduling of HPC applications on distributed cloud-oriented data centers. *Journal of Parallel and Distributed Computing* 2011; **71**(6):732–749.
27. Zikos S, Karatza HD. Performance and energy aware cluster-level scheduling of compute-intensive jobs with unknown service times. *Simulation Modelling Practice and Theory* 2011; **19**(1):239–250.
28. Bersani MM, Bianculli D, Dustdar S, Gambi A, Ghezzi C, Krstić S. Towards the formalization of properties of cloud-based elastic systems. *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems, PESOS 2014*. ACM: New York, NY, USA, 2014; 38–47.
29. Milojicic D, Llorente IM, Montero RS. OpenNebula: a cloud management tool. *Internet Computing, IEEE*. 2011; **15**(2):11–14.
30. Chen F, Grundy J, Schneider JG, Yang Y, He Q. Automated analysis of performance and energy consumption for cloud applications. *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14*. ACM: New York, NY, USA, 2014; 39–50.
31. Fargo F, Tunc C, Al-Nashif Y, Hariri S. Autonomic performance-per-watt management (APM) of cloud resources and services. *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC '13)*, ACM: New York, NY, USA, 2013; 2:1–2:10.
32. Chuang WC, Sang B, Yoo S, Gu R, Kulkarni M, Killian C. Eventwave: programming model and runtime support for tightly-coupled elastic cloud applications. *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*, ACM: New York, NY, USA, 2013; 21:1–21:16.
33. Hendrickson B. Computational science: emerging opportunities and challenges. *Journal of Physics: Conference Series*. 2009; **180**(1):1–8.
34. Lee Y, Avizienis R, Bishara A, Xia R, Lockhart D, Batten C, Asanovic K. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, ACM, New York, US, 2011; 129–140.
35. Cai B, Xu F, Ye F, Zhou W. Research and application of migrating legacy systems to the private cloud platform with cloudstack. *2012 IEEE International Conference on Automation and Logistics (ICAL)*, IEEE Computer Society Washington, DC, USA, 2012; 400–404.
36. Wen X, Gu G, Li Q, Gao Y, Zhang X. Comparison of open-source cloud management platforms: OpenStack and OpenNebula. *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2012; 2457–2461.
37. Baliga J, Ayre R, Hinton K, Tucker R. Green cloud computing: balancing energy in processing, storage, and transport. *Proceedings of the IEEE* 2011; **99**(1):149–167.
38. Spinner S, Kounev S, Zhu X, Lu L, Uysal M, Holler A, Griffith R. 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)*, IEEE Computer Society Washington, DC, USA, 2014; 157–166
39. Chiu D, Agrawal G. Evaluating caching and storage options on the amazon web services cloud. *2010 11th IEEE/ACM International Conference on Grid Computing (GRID)*, IEEE Computer Society Washington, DC, USA, 2010; 17–24.
40. Imai S, Chestna T, Varela CA. Elastic scalable cloud computing using application-level migration. *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC '12)*, IEEE Computer Society: Washington, DC, USA, 2012; 91–98.
41. Mastelic T, Oleksiak A, Claussen H, Brandic I, Pierson JM, Vasilakos AV. Cloud computing: survey on energy efficiency. *ACM Computer Surveys*. 2014; **47**(2):33:1–33:36.
42. Orgerie AC, Assuncao MDD, Lefevre L. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys* 2014; **46**(4):1–31.
43. Chen F, Grundy J, Schneider JG, Yang Y, He Q. Automated analysis of performance and energy consumption for cloud applications. *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14*. ACM: New York, NY, USA, 2014; 39–50.
44. Miettinen P, Vreeken J. MDL4BMF: Minimum description length for boolean matrix factorization. *ACM Transactions on Knowledge Discovery Data* 2014; **8**(4):18:1–18:31.
45. Tan L, Kothapalli S, Chen L, Hussaini O, Bissiri R, Chen Z. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Computing* 2014; **40**(10):559–573.
46. Padoin E, de Oliveira D, Velho P, Navaux P. Time-to-solution and energy-to-solution: a comparison between ARM and Xeon. *2012 Third Workshop on Applications for Multi-Core Architectures (WAMCA)*, IEEE Computer Society Washington, DC, USA, 2012:48–53.
47. Wilkinson B, Allen C. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson/Prentice Hall: Upper Saddle River, New Jersey, US, An Alan R Apt book, 2005.
48. Banas K, Kruzel F. Comparison of Xeon Phi and Kepler GPU performance for finite element numerical integration. *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), HPCC '14*. IEEE Computer Society: Washington, DC, USA, 2014; 145–148.
49. Hawick KA, Playne DP, Johnson MGB. Numerical precision and benchmarking very-high-order integration of particle dynamics on GPU accelerators. *Proceedings International Conference on Computer Design (CDES'11), CDE4469*. CSREA: Las Vegas, USA, 2011; 83–89.

DOI: 10.1002/cpe

50. Comanescu M. Implementation of time-varying observers used in direct field orientation of motor drives by trapezoidal integration. *6th IET International Conference on Power Electronics, Machines and Drives (PEMD 2012)*, IET, London, England, 2012; 1–6.
51. Tripodi E, Musolino A, Rizzo R, Raugi M. Numerical integration of coupled equations for high-speed electromechanical devices. *IEEE Transactions on Magnetics* 2015; **51**(3):1–4.
52. Islam S, Lee K, Fekete A, Liu A. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering (ICPE '12)*, ACM: New York, NY, USA, 2012; 85–96.
53. Mao M, Humphrey M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11*. ACM: New York, NY, USA, 2011; 49:1–49:12.
54. Zhang Y, Sun W, Inoguchi Y. Predict task running time in grid environments based on CPU load predictions. *Future Generation Computer Systems* 2008; **24**(6):489–497.

DOI: 10.1002/cpe