Towards providing middleware-level proactive resource reorganisation for elastic HPC applications in the cloud

Rodrigo da Rosa Righi*,
Vinicius Facco Rodrigues,
Igor Fontana de Nardin and
Cristiano André da Costa

Applied Computing Graduate Program,
Unisinos University,
São Leopoldo, Rio Grande do Sul, Brazil
Email: rrrighi@unisinos.br
Email: vfrodrigues@unisinos.br
Email: nardinif@unisinos.br
Email: cac@unisinos.br
*Corresponding author

Marco Antonio Zanata Alves

Federal University of Paraná – UFPR,
Curitiba, Paraná, Brazil
Email: mazalves@inf.ufpr.br

Maurício Aronne Pillon

Santa Catarina State University – UDESC,
Joinville, Santa Catarina, Brazil
Email: mauricio.pillon@udesc.br

Abstract: Elasticity is one of the most important features of cloud computing, referring to the ability to add or remove resources according to the needs of the application or service. Particularly for High Performance Computing (HPC), elasticity can provide a better use of resources and also a reduction in the execution time of applications. Today, we observe the emergence of proactive initiatives to handle the elasticity and HPC duet, but they present at least one problem related to the need of a previous user experience, large processing time or completion of parameters. Concerning the aforesaid context, this paper presents ProElastic – a lightweight model that uses proactive elasticity to drive resource reorganisation decisions for HPC applications. Our idea is to explore both performance and adaptivity at middleware level in an effortless way at user perspective. The results showed performance gains and a competitive cost (application time × consumed resources).

Keywords: cloud elasticity; proactive optimisation; performance; resource management; adaptivity.

Reference to this paper should be made as follows: da Rosa Righi, R., Rodrigues, V.F., de Nardin, I.F., da Costa, C.A., Alves, M.A.Z. and Pillon, M.A. (2019) 'Towards providing middleware-level proactive resource reorganisation for elastic HPC applications in the cloud', *Int. J. Grid and Utility Computing*, Vol. 10, No. 1, pp.76–92.

Biographical notes: Rodrigo da Rosa Righi is Assistant Professor and researcher at Unisinos. He obtained his PhD degree in Computer Science in 2005.

Vinicius Facco Rodrigues is a PhD student in Applied Computing and researcher at Unisinos University. His research areas are computer networks and distributed systems.

Igor Fontana de Nardin is graduated in Science Computer by Unisinos University. His research areas are computer networks and distributed systems.

Cristiano André da Costa is a Full Professor at Unisinos and he obtained his PhD degree in Computer Science at UFRGS in 2008.

Marco Antonio Zanata Alves is adjunct Professor at UFPR since 2016. He has experience in computer science, focusing on computer architecture area.

Maurício Aronne Pillon is Professor at UDESC since 2006. He has experience in computer networks, distributed systems and parallel systems.

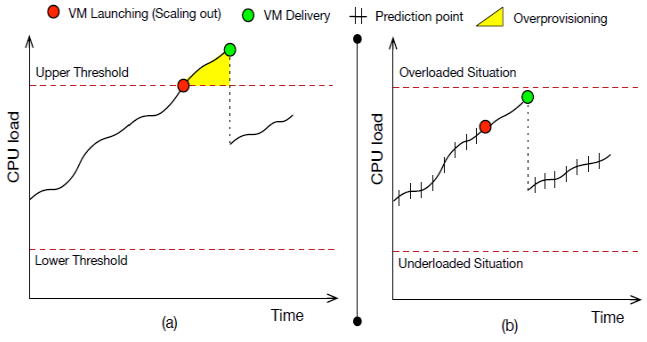
1 Introduction

Currently, there are many "Grand Challenge" problems that require fast responses, such as the weather forecast simulation that processes a large number of data to predict the climate situation for the next day. Running large and accurate simulations commonly require a large number of computing resources, demanding the use of supercomputers, computer clusters or grids (Weidner et al., 2016). Thus, scientific computing has historically been dependent on the advances of High Performance Computing (HPC) and parallel processing. In general, supercomputers, clusters and grids have a fixed number of resources that must be maintained in terms of infrastructure configuration, scheduling (where tools such as PBS[1], OAR[2], OGS[3] are usually employed for resource reservation and job scheduling) and energy consumption (Weidner et al., 2016; Lynar et al., 2011; Lynar et al., 2013; Lorido-Botrán et al., 2012; Harvey et al., 2016). In addition, optimising the number of processes to execute a HPC application can be a hard procedure: (i) both short and large values will not explore the distributed system in an efficient way; (ii) a fixed value cannot fit irregular applications, where the workload varies along the execution and/or it is not predictable in advance.

In addition to the aforementioned computing infrastructure sources, cloud computing has proved itself as a new way to acquire computing resources on demand (Lorido-Botrán et al., 2012; Beernaert et al., 2012; Gong et al., 2010). An important characteristic, not available on traditional architectures (e.g., clusters and grids) emerged on cloud computing: elasticity (Chilipirea et al., 2016; Netto et al., 2014; da Rosa Righi et al., 2015). Elasticity is defined as the ability of a system to dynamically add or remove computational resources used by either an application or user to match the current demand as closely as possible. This facility is gaining attention of the HPC community thanks to the benefits it can provide, that include improvements in performance, cost (normally obtained by multiplying application time by the number of resources) reduction and better resources utilisation. In theory, now the programmer does not need to take care of the number of processes and resources for the target application, since both will be adapted at runtime. However, as will be discussed later, this scenario is not trivial and depends on a series of factors including user experience and application and elasticity modelling (da Rosa Righi et al., 2015; Coutinho et al., 2016).

Today, most of the elasticity control strategies can be classified as being reactive or proactive (also named by some authors as predictive) (Galante et al., 2016) (see Figure 1). Reactive approaches are based on both static thresholds and if-condition-then rules to manage elasticity (Beernaert et al., 2012; da Rosa Righi et al., 2015). Although simple and intuitive, the task of completing these parameters is not trivial sometimes requiring deep knowledge about the behaviour of the system over time. This makes the accuracy of the policy subjective and prone to uncertainty: the same set of thresholds that fits fine a specific infrastructure/ application possibly causes undesired emergent behaviours, such as instability and resource thrashing, on other settings. In addition, the problem of using fixed thresholds is related to the lack of reactivity. There are situations in which the cloud controller could anticipate the (de)allocation of resources, but the resource configuration remains the same due to bad choices on setting the lower and upper load thresholds (Chilipirea et al., 2016; Coutinho et al., 2016; Galante et al., 2016).

Figure 1 Elasticity approaches: (a) reactive; (b) proactive



On the other hand, a proactive approach employs prediction techniques to anticipate the behaviour of the system (its load) and thereby decide the reconfiguration actions (Gong et al., 2010; Stelmar Netto et al., 2014; Rosa et al., 2014). To accomplish this approach, it is common to use machine learning algorithms including Neural Network, Linear Regression, Support Vector Machine, Reinforcement Learning and Pattern Matching techniques (Galante et al., 2016). Although not needing thresholds, this approach is normally based on a robust mathematical modelling, being classified adversely as time-consuming for sensitive performance-driven applications (Palacin et al., 2016). There is also the need of training the predictive technique at design-time and previous execution of the application to optimise the selection of parameters (Galante et al., 2016; Rosa et al., 2014). Finally, Netto et al. (2014) affirm that proactive elasticity strategies focus only on method accuracy and ignore cloud technical limitations such as the time of a scaling up operation, besides being very much dependent on workload characteristics.

In our understanding, at user side, proactive elasticity is better because it does not need to complete thresholds or rules as the reactive approach does. However, the state-of-

the-art in proactive elasticity concerns at least one problem related to prior user experience on using elasticity, changes in the application code, time-consuming approach, stop-reconfigure-and-go and need of a previous execution to tune parameters or prediction models (Gong et al., 2010; Netto et al., 2014; Rosa et al., 2014; Perez-Palacin et al., 2016; Roy et al. 2011). Considering this background, we propose ProElastic – a lightweight proactive elasticity model that provides resource reorganisation for iterative HPC applications. The lightweight characteristic is explained in two ways: first, we are using ARIMA-based time series to predict the application behaviour by monitoring CPU load of virtual resources, which computes fast and does not need any previous execution of the application (Kalpakis et al., 2001; El Hag and Sharif, 2007; Masood and Schmidt, 2015); second, the design of loosely-coupled architecture allows the execution of scaling out operations in parallel with the HPC application, so not blocking the application when reconfigurations are in course. In addition, ProElastic acts at the PaaS (Platform as a Service) level of a cloud, not imposing any change in the application code to take profit from resource reorganisation. Effortlessly, the programmer only compiles the application with ProElastic middleware, which transforms a non-elastic application in an elastic one.

Our final goal is not only to get a better execution time, but also an equal or better cost (consumed resources × performance) when comparing ProElastic against non-elastic and reactive elasticity managers. In Figure 1(b), we illustrate the ProElastic's idea on adding new resources earlier, so reducing the application time and not executing in over- or under-loaded situations. Based on the proposed model, we developed and evaluated a prototype that executes a master-slave iterative application over a private cloud built over OpenNebula. This evaluation also considered four pertinent input workloads to analyse the aforementioned metrics under different load patterns. Our evaluations followed the principles defended by Isalm et al. (2012), who argued that the use of synthetic workloads is considered as a representative form to evaluate elasticity in computational clouds. The results were promising, emphasising the accuracy of the predictions which were reflected in the values of application time and cost in favour of ProElastic. The main scientific contributions of this paper are the following:

- The ProElastic framework, which not only transforms a non-elastic application in an elastic one but also presents a communication architecture between applications processes and the elasticity manager.
- We modelled a framework to enable a novel feature denoted asynchronous elasticity, where VM (Virtual Machine) transferring or consolidation happens in parallel with application execution.

The remainder of this article will first introduce related work in Section 2. Section 3 describes ProElastic, its architecture and algorithms to control elasticity proactively. A ProElastic prototype is presented in Section 4. Evaluation methodology and a discussion of the results are presented in Sections 5 and 6, respectively. Finally, Section 7 emphasises the scientific contribution of the work and notes several challenges that we can address in the future.

2 Related work

This section describes some approaches to manage elasticity in cloud. They were divided into two groups: reactive managers in Subsection 2.1 and proactive managers in Subsection 2.2. Lastly, the initiatives were compared and analysed in order to detach the current gaps in the cloud elasticity research area.

2.1 Reactive managers

Reactive managers are those based only on thresholds to take elasticity decisions; more precisely, resource reconfiguration takes place when the lower or the upper threshold is violated. The advantage of this model is its simplicity on both decision making and information collection, but new resources will not be available when they are really needed because scaling out operations are normally time-consuming. Reactive elasticity is mainly explored on transactional and Web applications that execute on public clouds like Amazon EC2 and Windows Azure (Albonico et al., 2016). In the HPC scope, we highlight two reactive elasticity initiatives: AutoElastic (da Rosa Righi et al., 2015) and Elastack (Beernaert et al., 2012). AutoElastic acts at the PaaS level of cloud, managing elasticity for master-slave applications where the users do not need to change any line in their applications to take profit from runtime resource reconfiguration. The user must define an SLA (Service Level Agreement) which contains the minimum and the maximum number of VMs that could be allocated to run the application. AutoElastic uses reactive and horizontal elasticity, so an action is taken when a threshold is reached to add or remove virtual machines.

Elastack (Beernaert et al., 2012) has an elasticity manager tied at IaaS (Infrastructure as a Service) level of the OpenStack middleware. This manager aims to enable adaptability and system monitoring, so automatically managing the resources of the cloud. Elastack architecture is divided into three components: Daemon Monitor, Controller Daemon and Serpentine Script. The monitor is a background process that runs on each compute node, updating the manager with the machines information. The controller is responsible for adding and removing resources from the cloud according to elasticity decisions. Finally, the serpentine script is the module that decides when elasticity takes place or not.

2.2 Proactive managers

Proactive managers, unlike reactive ones, try to predict the cloud behaviour to anticipate elasticity decisions before any under or overload situation. When the system identifies a future requirement, an elasticity action is immediately taken;

so when a resource is required, it is already available to the application processes. In general, the managers here are divided into two groups, which are composed considering the used prediction algorithm: machine learning and statistical model (Rosa et al., 2014). Machine learning algorithms need more cycles of initialisation before starting the prediction procedures than the statistical models (Rosa et al., 2014).

Rosa et al. (2014) presented an experimental tool to predict workloads in clouds. This tool was created to assess the demands of server resources, using historical data of the application to predict when more resources are needed. To predict the workloads, the authors use two different models: Support Vector Regression and Naive Bayes. These two machine-learning-based algorithms usually need some learning cycles to start the prediction activity. This implementation was developed using Amazon cloud and its Java API.

Vadara (Loff and Garcia, 2014) is a framework for managing elasticity which is not tied to any cloud provider. The approach of elasticity is predictive, where the system tries to predict workloads. Data is recorded along the time, allowing it to be applied on predictive algorithms. To predict the application workload, Vadara uses k-Nearest Neighbours (kNN) to combine the generic elasticity strategies. With this algorithm, you can select the best technique according to the latest workloads. In the tests, the Holt-Winters, ARIMA and StructTS methods were used, so that the kNN can choose the best for a particular workload.

The Insights Platform (Moore et al., 2013) is a framework which offers a real-time cloud performance monitor that is in charge of elasticity decisions. The main idea of the authors is to transform a reactive cloud in a proactive cloud. Both algorithms, reactive and proactive, are used to enhance performance in the cloud, enabling then one controller for each mode. The predictive control was created using the Weka machine learning library, comprising three prediction models: one based on time series and two based on Naive Bayes.

Barrett et al. (2013) present an elasticity model using the Q-learning concept. This model aims to determine the best resources for the current workload using Xen Hypervisor which is a popular open source virtualisation framework that can work on various cloud providers. The proposal has an architecture where the user does not have to worry about managing the cloud, simply entering an SLA with the application requirements. Each agent makes decisions to add, remove or keep the VMs allocated to the application. These actions are taken according to different variables, such as cost of the resource, penalty related to an SLA violation, CPU used, memory usage, etc. With this architecture, the system decides the best moment of changing a VM capacity using the Amazon cost function for each type of virtual machine.

Nikravesh et al. (2015) investigated the best algorithm between Support Vector Machine (SVM) and Neural Networks (NN) for prediction of workloads. To accomplish this, they developed different elasticity managers considering the two aforementioned algorithms, testing them with a predefined set of workloads. The authors used a metric named workload performance (user requests per unit of time). The final aim was to compare the prediction accuracy of the two algorithms for different patterns of workloads.

Gong et al. (2010) present a cloud resource management model that uses statistical models to predict the load and take elasticity decisions. The article presents a manager called PRESS and compares its performance with others algorithms, which were implemented in the Xen platform. The authors modelled PRESS with two prediction algorithms. The first intends to be applied to cyclical loads using the signal processing technique called Fast Fourier Transform. For loads of non-cyclical work, it is used Markov chains. This model requires some boot cycles to start the prediction machine.

Roy et al. (2011) describe a resource allocation algorithm based on predictive models, trying to predict future workloads in order to reduce the application cost. The cost is given by some factors such as the SLA violations, rental cost of resources and the cost associated with the configuration changes. To be able to predict future workloads, the authors used the autoregressive moving average (ARMA) approach. In addition to the ARMA, the authors used Mean Value Analysis to identify the bottlenecks in the application, where one of the parameters of this algorithm is the prediction given by ARMA. Finally, it is calculated the cost of making changes in the cloud. This is necessary because each change in the cloud has a cost associated to it, so it is advantageous to know the cost and when it is the best time to apply a particular change.

2.3 Analysis and research opportunities

Table 1 presents a comparison among the initiatives discussed in this section. Considering the last column, we observe that there are two main application models in the cloud: (i) transactional, which are accessed through the Web and execute in a request-reply interaction pattern (such as e-commerce) and; (ii) batch, in which a user launches a request and expects its results (such as data mining, graphics rendering and scientific application demands) (Martineau et al., 2016). Table 1 shows that all work that provide proactive elasticity cover transactional applications, where metrics like cost and system throughput are prioritised. Only two works (Beernaert et al., 2012; da Rosa Righi et al., 2015) were built to improve batch applications, but they do not provide any kind of proactiveness on resource management. We also observe that most of those works (Gong et al., 2010; Rosa et al., 2014; Roy et al., 2011; Barrett et al., 2013) focus on reducing cloud costs. In terms of transactional applications submitted to the cloud this is a major concern, since users are usually subjected to pay for using the cloud, like Amazon EC2, and every hour (the charge unit varies among the providers) of allocation is charged. On the other hand, the main objective on executing batch applications is performance, because of these applications normally represent CPU- and/or IO-intensive problems that require distributed resources and long-running demands.

Table 1 Related work comparison

Authors	*Elasticity*	*Prediction algorithm*	*Focus manager*	*Application Model*
Da Rosa Righi et al. (2015) (AutoElastic)	Reactive	–	High-performance	Batch, master-slave iterative
Beernaert et al. (2012) (Elastack)	Reactive	–	Automating elasticity	Batch, no application model
Rosa et al. (2014)	Proactive	Machine learning (SVR and Naïve Bayes)	Avoid cloud oversizing	Transactional, web requests
Gong et al. (2010) (PRESS)	Proactive	Time Series (Fast Fourier transform and Markov chain)	Reduce cost and SLO violations	Transactional, web requests
Loff and Garcia (2014)	Proactive	Machine learning (kNN)	Manager uncoupled from the provider	Transactional, web requests
Roy et al. (2011)	Proactive	Time Series (ARMA)	Reduce cost	Transactional, web requests
Moore et al. (2013) (Platform Insights)	Proactive	Machine learning (Time series and Naïve Bayes)	Transform reactive cloud in proactive	Transactional, web requests
Barrett et al. (2013)	Proactive	Machine learning (Q-Learning)	Optimise cloud resources	Transactional, web requests
Nikravesh et al. (2015)	Proactive	Machine learning(SVM e NN)	Predict workload	Transactional, web requests

Comparing the proactive elasticity managers, we observe that machine learning techniques are used in most of the initiatives (Rosa et al., 2014; Loff and Garcia, 2014; Moore et al., 2013; Barrett et al., 2013; Nikravesh et al., 2015). Given that these managers are in charge of supporting transactional applications, the use of machine learning seems the most appropriate method here because most of the prediction problems is to identify request patterns to web servers. Chang (2016) affirms that sometimes, or yet in most of the times, machine learning will fail, thus it requires some understanding of the problem beforehand in order to apply the right parameters. Machine learning-based proactiveness requires a startup period greater than the statistical models, which is not a critical problem for web applications, but can become an issue for batch demands (Chang, 2016). In addition, machine learning techniques are normally heavier in terms of CPU cycles than the statistical model, so it is common to employ clusters and/or GPU boards to execute them efficiently.

AutoElastic (da Rosa Righi, 2015) and Elastack (Beernaert et al., 2012) initiatives aim to increase performance for batch applications, without any feature related to proactivity, but using thresholds to accomplish elasticity. Other proposals (Gong et al., 2010; Rosa et al., 2014; Roy et al., 2011; Loff and Garcia, 2014; Moore, 2013; Barrett et al., 2013; Nikravesh et al., 2015) operate proactively and seek to improve application performance, but are employed on transactional applications. This kind of demand only presents a request-reply interaction, where a manager: (i) receives input requests; (ii) manages load balancing and elasticity and; (iii) dispatches requests and receives replies to/from processing replicas. The same does not occur on batch applications, where data and control dependencies and inter-process communication (i.e., communication among the replicas) can take place. Finally, we identified a gap on developing an elasticity manager that combines: (i) lightweight proactive elasticity; (ii) focus on HPC applications; (iii) performance (both in terms of application time and prediction calculus) and; (iv) the interference of the user/programmer must be as little as possible.

3 ProElastic proposal

This section presents the ProElastic model, describing firstly its general ideas and, secondly, its architecture and elasticity model.

3.1 ProElastic principles

ProElastic is a proactive elasticity model that aims to improve performance, but not neglecting the cost (here denoted as performance × used resources), for iterative master-slave applications that run in the cloud. Our idea is to provide proactive elasticity in a transparent and effortless way at user viewpoint, who does not need to write rules and actions for resource reconfiguration as required in reactive approaches. In addition, users must not need to change their parallel application, so not inserting any elasticity calls from a particular library or modifying the application to add/remove resources by themselves. Acting at the PaaS level of a cloud, ProElastic firstly transforms a non-elastic application in an elastic one and secondly, it manages resource (and also application processes, consequently) reorganisation through automatic VM allocation and consolidation procedures. The proposal must be aware of the VM instantiation overhead to provide seamless elasticity, i.e., in a non-prohibitive way for HPC applications.

Figure 2 General ideas on using elasticity: (a) standard approach adopted by Amazon AWS and Windows Azure, in which the user must pre-configure a set of elasticity rules and actions; (b) Prolastic idea, contemplating a manager that coordinates the elasticity actions and configurations on behalf of the user

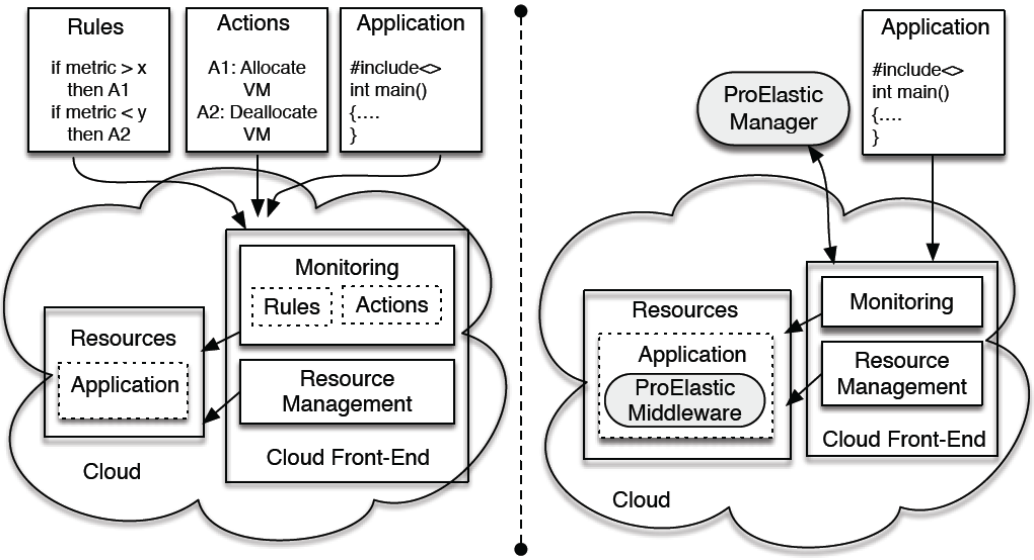


Figure 2(a) illustrates the traditional approaches of providing cloud elasticity to HPC applications, while part (b) highlights ProElastic's idea. We are offering a middleware in which user must compile his/her application with, and also a manager that controls resource reorganisation. The proactive nature of the model refers to not only the interactionless behaviour at user viewpoint, but also the capacity to anticipate elasticity actions based on historical data. To accomplish this, we are using the statistical model to provide predictions; more precisely, ARIMA-based (Kalpakis et al., 2001; El Hag and Sharif, 2007; Masood and Schimidt, 2015) time-series were used to compute the load metric for the application. Other prediction algorithms like ARMA and Moving Average (MA) were passed over because they present at least one of the problems: demonstrate a not accurate prediction; time-consuming; many boot cycles to start predictions. Taking into account our focus on CPU-bound HPC applications, ProElastic works with the CPU metric of the virtual machines in the time series calculus.

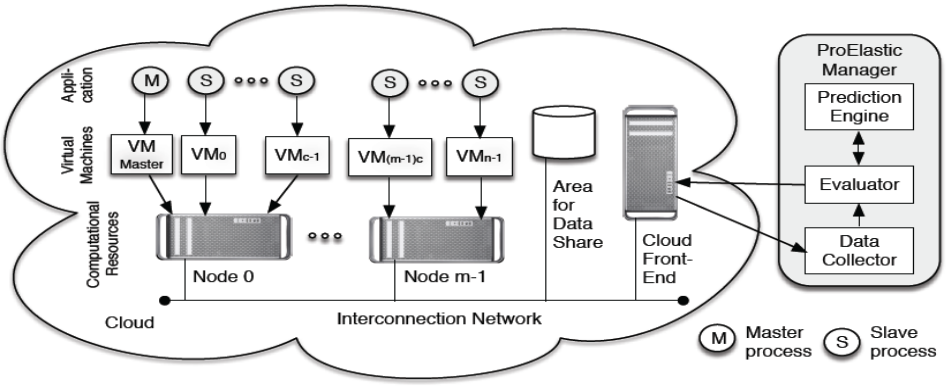
We are focusing on master-slave iterative applications, i.e., applications that are characterised as a collection of loops. Although trivial, this style is used in several areas, such as genetic algorithms, Monte Carlo techniques, geometric transformations in computer graphics, SETI@Home-like applications, cryptography algorithms and applications that follow the Embarrassingly Parallel computing model (Martineau et al., 2016; Raveendran et al., 2011). Particularly, the iterative nature is pertinent to our purposes for the following rationale: (i) unlike using application time, the use of the number of loop refers to a significative meaning in the prediction calculus; (ii) either the beginning or the end of loop can be a pertinent part to insert elasticity code; (iii) in the beginning of a loop, we have a consistent global state of the distributed system in which there are not in-transit communications.

3.2 Architecture

Figure 3 shows the ProElastic architecture, highlighting both the elasticity manager and cloud components. ProElastic employs horizontal elasticity in which the work granularity is a node with *c* virtual machines, where *c* denotes the number of processing cores. Each virtual machine refers to a slave process that runs solely to exploit the full power of a single core. We are neglecting the use of vertical elasticity (i.e., VM resizing; Galante et al., 2016) because this approach is limited to the processing power of a single node. In addition, VM resizing normally implies in the stop-reconfigure-and-go approach, which is prohibitive for time sensitive applications like the HPC ones. Thus, horizontal approach is useful for enabling resource deallocation and for enlarging the infrastructure beyond the limits of a single resource.

The most important module of the ProElastic architecture is the Prediction Engine, which is in charge of predicting load values based on the CPU metric. The manager uses the API offered by the cloud provider to both collect monitoring data and to trigger elasticity actions. The Evaluator module receives the predicted load from the Prediction Engine and uses this data on elasticity decision making. If a reconfiguration is required, the manager sends a notification to the master process. The communication between the manager and the master process is performed using a shared data 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 (Vozmediano et al., 2012; Cai et al., 2012). ProElastic manager uses SSH to login in the cloud front-end and to read/write from/to the shared area afterwards.

Figure 3 ProElastic 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 $c \times m$

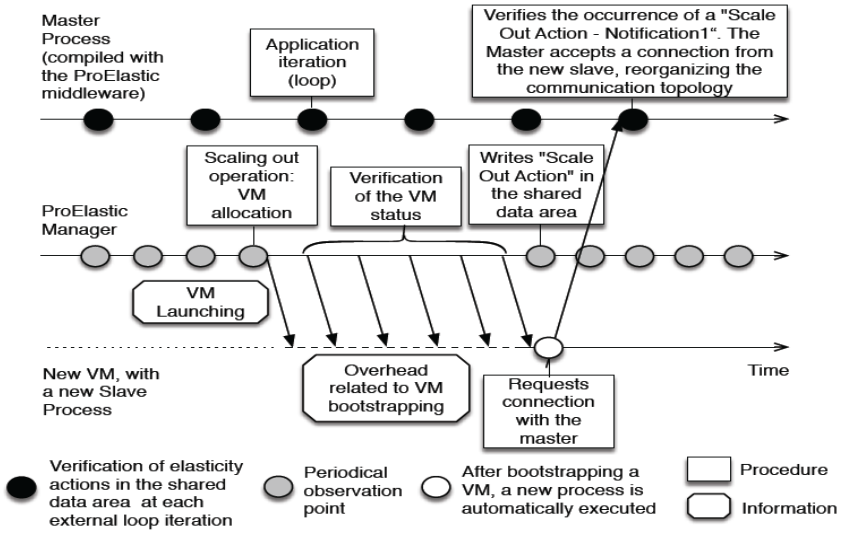


Concerning the focus on providing a lightweight proactive approach, ProElastic uses ARIMA-based time series which is quickly computed only using a single node; in our case, in the manager. In addition, the lightweight term is also related to the design of the proposed model. Instead of offering an application-sided elasticity, the use of a manager brings the non-blocking benefit at the application perspective when resource reconfigurations take place. We named this feature as asynchronous elasticity, in which the application is notified as soon as a new computing VM instance (scale out) is available in the system without impairing its normal execution flow. However, this non-blocking operation implies in the following question: How can we notify the application about the resource reconfiguration? We used the shared data area for this purpose. At each loop iteration, the master verifies in the shared data area whether the elasticity manager signalises the existence of resources to add or drop to/from the communication topology. In detail, ProElastic enables three notifications:

- The manager writes to the shared area, whereas application processes read from it:
  - Notification 1: there is a new compute node with c VMs, each one with a new application process that has an IP and a unique identification.
  - Notification 2: 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:
  - Notification 3: this gives permission to consolidate the previously requested node.

Based on Notification 1, the current processes may start working with the new set of resources (a single node with c VMs, each one with a new process). Figure 4 illustrates the functioning of the ProElastic Manager when creating a new slave, so launching Notification1 afterwards. Notification 2 is relevant for the following reasons: (i) not stopping a process executing while either communication or computation procedures take place; (ii) ensuring that application will not be aborted with the sudden interruption of one or more processes. In particular, the second reason is important for MPI (Message Passing Interface) applications that run over TCP/IP networks, since they commonly crash with a premature termination of any process (Galante et al. 2016). Notification 3 is normally taken by a master process, which ensures that the application has a consistent global state where processes may be disconnected properly. Afterwards, the remaining processes do not exchange any message to the given node.

Figure 4 Functioning of the master, the new slave and the ProElastic Manager to enable the asynchronous elasticity



3.3 Elasticity management and prediction model

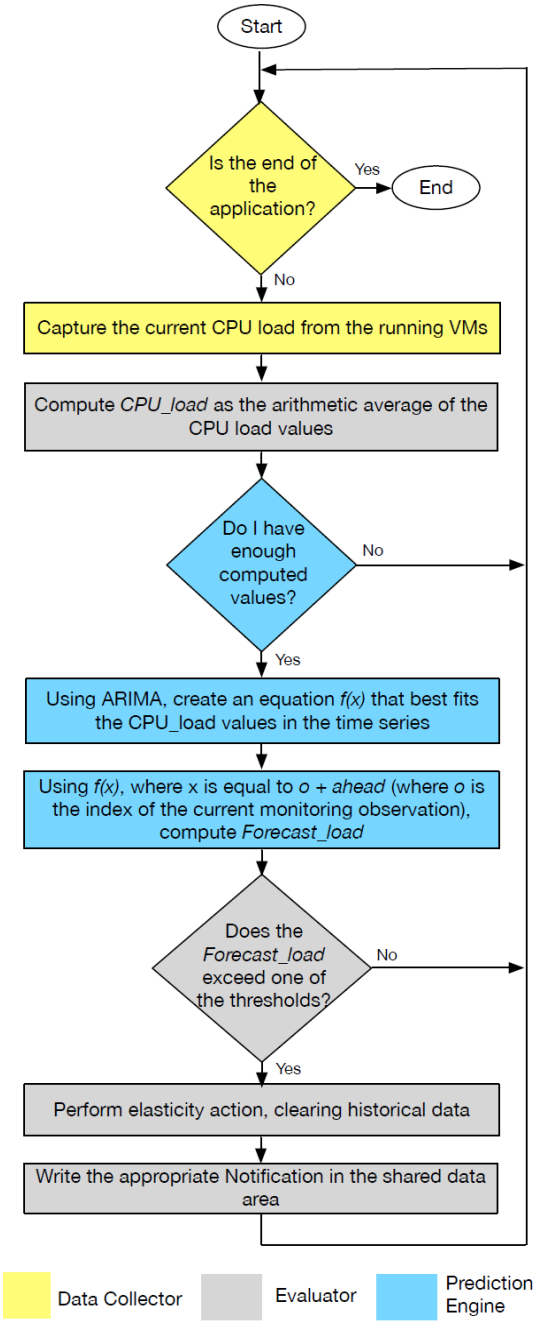
ProElastic addresses scaling in and out operations in such a way that elasticity actions are anticipated, so resources are completely delivered earlier than reaching either an under- or over-provisioned situation. Since ProElastic performs periodical monitoring, elasticity can occur only in discrete observation points where cloud data collection and elasticity decision making are activated. Evaluator and Prediction Engine modules are in charge of managing elasticity actions. Their interactions can be observed in the flowchart of Figure 5, which depicts the ProElastic's periodical monitoring loop. Below we detail the responsibility of each module:

- *Evaluator*: This modules act in three moments. First, when a monitoring observation takes place, Data Collector module captures all CPU loads from all running VMs. Evaluator takes this data to compute *CPU_load* as the arithmetic average of the load values. Second, *CPU_load* is sent to the Prediction Engine module which performs a series of calculus, so returning *Forecast_load* back to the Evaluator afterwards. Third, to decide about elasticity actions, the Evaluator must decide if *Forecast_load* indicates an overloaded or underloaded situation. Considering related work (Netto et al., 2014; da Rosa Righi et al., 2015; Galante et al., 2016), we are using lower and upper load limits for this; particularly, the values of 20% and 80% of CPU load were adopted. If one of these limits is exceeded, a resource reconfiguration takes place to address such a situation. Our work grain is a single compute node. A scaling out operation involves the addition of a node and c VMs (where c denotes the number of processing cores of the node), while a scaling in operation turns off a particular node and all VMs running on it.

- *Prediction Engine*: After receiving *CPU_load* relative to the o^{th} monitoring observation, Prediction Engine stores this value locally and performs two operations: (i) development of a regression equation that better fits all stored *CPU_load* values from now up the last VM delivery or up to the beginning of the application; (ii) using the aforesaid equation, the idea is to determine the system load in the future. To accomplish (i), we configure ARIMA to act as the Holt-Winters method (El Hag and Sharif, 2007), so considering the Triple Exponential Smoothing approach to propose the equation. This method assigns exponentially decreasing weights as the observation get older. Taking into account this equation in the $f(x)$ style, in (ii) we must decide x as the moment that we would like to predict the CPU load. In other words, considering that we know the number of o, we must decide *ahead* since o+ *ahead* will serve as x in the equation. To compute *ahead,* we are using equation (1). Equation (1) considers the maximum value among all times regarding already performed scaling out operations, which is feasible thanks to always using the same VM template that represents a slave process. Finally, the forecasted value is assigned to *Forecast_load*, which is sent to the Evaluator module.

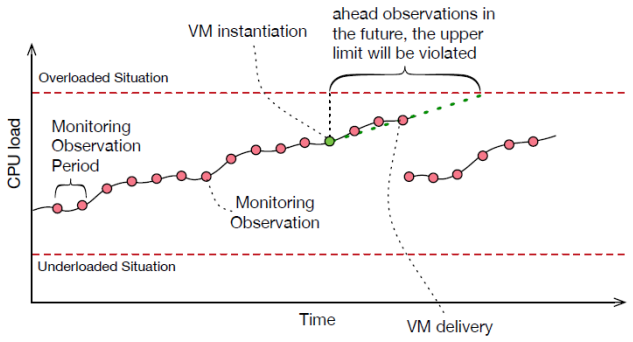
$$ahead = \frac{Abs(Max(Scaling\_out\_time))}{monitoring\_observation\_period} \quad (1)$$

Figure 5 Flowchart of the ProElastic Manager periodical monitoring loop. First, the Data Collector module collects CPU load values to compute the total load. It needs a collection of values to start the Prediction Engine. Second, the Prediction Engine uses this collection to create an equation, so using it to verify that a threshold will be violated when crossing n loops ahead in the execution. Finally, elasticity actions take place if the Evaluator detects the need to add or remove resources



The variable *ahead* points out as soon as an elasticity action must be done. Equation (1) shows how we compute this value, preventing us from two pitfalls: (i) if *ahead* is considered as very large, the prediction method can miss the next short-term steps of the application behaviour so incurring in an eventual false-positive or false-negative elasticity situation; (ii) if *ahead* is considered as too short, an elasticity action can deliver the resources after its real need by the application, so incurring in an overloaded situation and performance loss. In equation (1), *Scaling_out_time* considers the VM instantiation time which involves the transferring of a VM template that will run a slave process and the complete bootstrap of the operating system. Figure 6 illustrates an example in which an elasticity action is taken because of ProElastic concluded that *ahead* monitoring observations in advance the system would execute in an overloaded state.

Figure 6 Example of elasticity prediction. In the green ball, ProElastic predicts a future value of CPU, so a new VM is instantiated because the value of the forecast CPU exceeds the upper threshold. This VM is ready for use after a few cycles then reducing the CPU load thanks to a better load balance among a greater number of resources



ARIMA was chosen because it is a widely used model in several areas that need any kind of time series-based prediction (El Hag and Sharif, 2007; Masood and Schmidt, 2015). The algorithm only begins to predict after a few boot cycles (fewer cycles than those needed in the machine learning model) (Kalpakis et al., 2001). Based on (Kalpakis et al., 2001), we are using 5 for this parameter, so $5 \times$ *monitoring_observation_period* denotes the time in the beginning of the execution where none resource reconfiguration takes place. In addition, this time is also considered when delivering a new VM, since historical data regarding *CPU_load* values is reset so new values must be collected to restart the prediction engine. In the rest of the article, we also reference this moment as warm-up prediction period.

3.4 Application model

ProElastic 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. ProElastic 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 several contexts. However, we emphasise 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 the support, for example, of bulk synchronous parallel and pipeline programming models (Zomaya et al., 1996). The ProElastic parallel applications follow the multiple program multiple data (MPMD) principle (Zomaya et al., 1996), in which master and slave processes have different executable codes; each is mapped to a different VM template. The intent is to offer application decoupling for processes with different purposes, enabling readability and facilitating the implementation of elasticity.

Figures 7(a) and (b) present pseudocode of a ProElastic-supported iterative application. The master code executes a series of tasks, capturing each one sequentially and parallelising one-by-one to be processed by slave processes. Figure 8 illustrates the synchronous functioning of the iterative application with a malleable number of slave processes. ProElastic works with the following MPI 2.0-like communications directives, as highlighted in Figure 7: (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 the master process launches the slaves using a spawn-like directive, the proposed model operates according to another approach of MPI 2.0 for dynamic process management: connection-oriented communication using point-to-point, as sockets do. The launching of a VM automatically occurs in the execution of a slave process, which requests a connection with the master afterwards. Here, we emphasise that an application with ProElastic does not need to follow the MPI 2.0 interface, but the semantic of each aforementioned directive.

The transformation of a non-elastic application into an elastic one can be modelled 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 7(a) can be developed for procedural languages. Independent of the strategy, the code required for elasticity is simple, as shown in Figure 7(c). First, we must verify whether there is a new action from the ProElastic manager in the shared data area. If Notification 1 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 Notification 2, the master removes from its group the processes that belong to a specific node. After doing that, it triggers Notification 3.

Figure 7 (a) and (b): multiple program multiple data-like parallel application model supported by ProElastic; and (c) elasticity code to be inserted transparently by ProElastic in the code of the Master when considering the user viewpoint

(a)

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

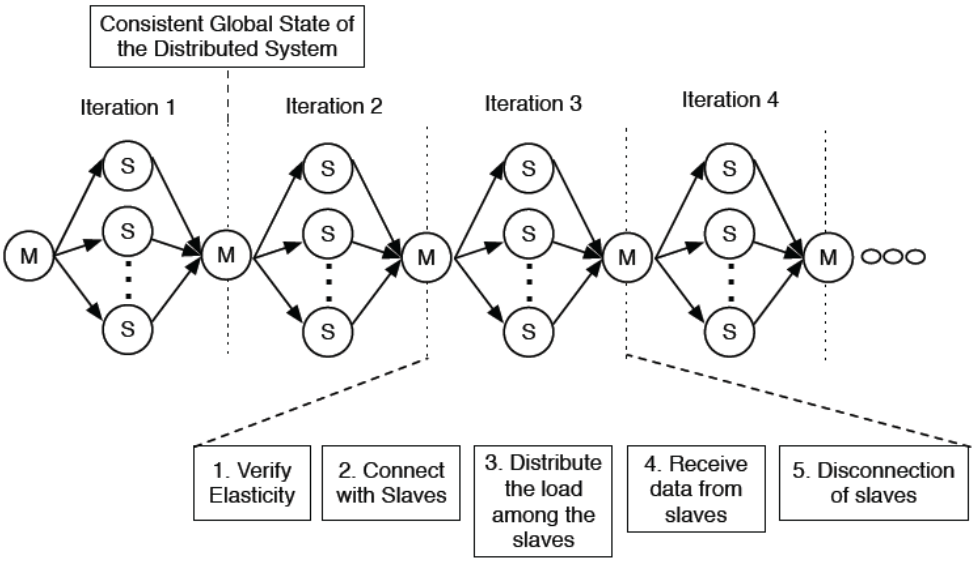
(b)

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

(c)

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

Figure 8 Application execution considering a malleable number of slave processes



Although the design of ProElastic considers master-slave applications, the iterative modelling 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 to optimise connection and disconnection procedures if a particular slave process remains active in the process list. This improvement can avoid too many TCP connections that require a three-way handshake protocol, which might be expensive for some applications.

4 Prototype implementation

This section describes the implementation decisions on writing a ProElastic prototype. We are using the OpenNebula (Moreno-Vozmediano et al., 2012) private cloud package to assembly our cloud environment. In addition, its Java API was also used to implement VM data monitoring and scaling in and out operations. The cloud configuration consists of eleven computers: ten of them are used as computing nodes and one acts as cloud front-end. All computers have the same configuration, which includes a Core 2 Duo E7500 2.9GHz processor and 4GB of RAM. The network interface cards are Gigabit Ethernet, however the switch is Fast Ethernet.

To implement the prediction algorithm, ARIMA, we used the JRI library that communicates with the R language. We opted to work with R because it already has a native library to work with ARIMA. The requirement here is that we need to install R on the same machine that ProElastic will operates. Tests conducted to evaluate the performance

of the Java with R integration demonstrated a very low overhead: the prediction engine takes about 500 milliseconds to return a response of a future metric when 20 values are considered in the time series. In the tests, each observation lasts approximately 15 seconds, so the aforesaid prediction time is not a performance problem.

The ARIMA has several configurations, based on three variables: p is the number of autoregressive terms, d is the number of differences and q is the number of terms in the moving average. The configuration used in this work considered the ARIMA with the following values: $p = 0$, $d = 2$ and $q = 1$. This configuration performs a triple exponential smoothing. Several ProElastic tests with different configuration possibilities for ARIMA were conducted and the aforementioned configuration was chosen since achieved the best performance.

5 Evaluation methodology

We developed a master-slave application that computes the numerical integration of a series of equations. The application follows the Newton-Cotes postulate that considers the area inside a closed interval by computing the area of several trapezoids (da Rosa Righi et al., 2015). The larger the number of subintervals, i.e. the number of trapezoids, the larger the CPU cycles to compute the application. The master reads a file containing a series of equations, so each loop is in charge of computing a single equation by passing a set of subintervals and the input equation for the existing slave processes. Aiming at observing ProElastic under different load situations, we designed four workloads: constant, ascending, descending and wave. Table 2 presents how we are computing them. We consider *load*(x) as the number of sub-intervals that will be calculated at each application iteration. For example, the Constant load maintains the number of processed subintervals as being the same for all equations, while in the descending load the application starts with a large number of subintervals, which becomes decrescent along the time.

Using the aforesaid application and the proposed workloads, we also modelled three scenarios to evaluate ProElastic:

- *Scenario 1* (*Non-Elastic*) considers the execution of the master-slave application in a non-elastic fashion, i.e., with a fixed number of processes.
- *Scenario 2* (*AutoElastic*) was developed to test the execution of the application when considering a reactive-based elasticity manager named AutoElastic. As ProElastic, AutoElastic also considers the CPU load as main metric on resource reorganisation decision making. Based on (da Rosa Righi et al., 2015), AutoElastic was configured with the values of 20% and 80% for the lower and upper thresholds, respectively.
- *Scenario 3* (*ProElastic*) presents the performance of the master-slave application which was compiled and executed with ProElastic.

Besides the performance perspective that represents final application time, we also compute the energy and the cost metrics. Particularly, energy consumption in computing is an important issue due to the increasing energy cost (Lynar et al., 2011; Lynar et al., 2013). Thus, we investigate the energy consumption by measuring resource allocation. Here, energy refers to the resource consumption which is rigid in Scenario 1 and malleable in Scenarios 2 and 3. To estimate the energy of the application, it is necessary to know the time that each deployment of virtual machines took place (see equation 2). In this equation, n means the maximum number of allocated VMs and $T(i)$ is the time spent when running a configuration with i VMs. For example, consider the situation: 20s with 2VMs, 120s with 4VMs, 100s with 6VMs and 80s with 4VMs; here we have $energy = 20 \times 2 + (120 \times 4 + 80 \times 4) + 100 \times 6 = 1440$. The cost, in its turn, is used to analyse how effective is the execution, with or without elasticity. Based on the standard notion of cost in the parallel computing area, which considers time and processors, here we use time and the previously computed energy metric (see equation 3). In this equation, $Time_{app}$ refers to the total time to run the application. Finally, since ProElastic is a proactive manager we are also analysing the accuracy of the prediction algorithm, comparing what was predicted against the actual CPU values.

$$Energy = \sum_{i=1}^{n} (i \times T(i)) \tag{2}$$

$$Cost = Energy \times Time_{app} \tag{3}$$

Table 2 Functions that define the input workloads

Load	*Load Function*	*Parameters*			
		v	*w*	*t*	*z*
Constant	$\text{load}(x) = w \div 2$	–	1,000,000	–	–
Ascending	$\text{load}(x) = x \times t \times z$	–	–	0.2	500
Descending	$\text{load}(x) = w - (x \times t \times z)$	–	1,000,000	0.2	500
Wave	$\text{load}(x) = v \times z \times sen(t \times z) + v \times z + w$	1	500	0.00125	500,000

6 Results

This section describes the results, which were organised in two subsections: (i) Subsection 6.1 presents the values of time, energy and cost for the three considered scenarios; (ii) Subsection 6.2 highlights some ProElastic executions, showing resource allocation and both predicted and actual CPU loads along the time.

6.1 Analysing time, energy and cost metrics

Table 3 shows the application time in seconds when considering the four workloads and three scenarios. The non-elastic execution was tested with 1, 2, 3 and 4 nodes. As detailed in Section 4, each node executes 2 VMs, each VM with a single slave process. We also present different initial configurations for AutoElastic and ProElastic. Table 3 shows that ProElastic is faster than AutoElastic, being also the best option in the most of the cases when comparing scenarios Non-Elastic and ProElastic. We observe that the initial configuration is crucial for performance purposes. For example, the execution with 4 fixed nodes presents a better performance when compared with elastic executions that started with 1 or 2 nodes. In this case, even enlarging the number of resources in AutoElastic and ProElastic, the application is not long-running enough to outperform the execution that started from a larger number of compute nodes. However, the larger the resource allocation along the time, the larger the energy metric as presented in Table 4. It is clear that time and energy are inversely proportional metrics. This explains the higher indexes for ProElastic in Table 4.

Table 3 Application time in seconds

		Workload			
	Nodes	*Constant*	*Wave*	*Ascending*	*Decreasing*
Non-Elastic	1	4043	4068	4026	4079
	2	2482	2574	2511	2531
	3	1736	1842	1765	1907
	4	1610	1701	1706	1651
AutoElastic	1	2118	2340	2179	2044
	2	2356	1843	2122	1732
	3	1756	1768	2092	1616
ProElastic	1	1821	2034	1673	1880
	2	1627	1752	1527	1675
	3	1550	1660	1524	1516

Table 4 Energy consumption in accordance with equation (2)

		Workload			
	Nodes	*Constant*	*Wave*	*Ascending*	*Decreasing*
Non-Elastic	1	8086	8135	8052	8158
	2	9928	10,297	10045	10125
	3	10417	11051	10590	11444
	4	12876	13610	13649	13205
AutoElastic	1	10578	12185	10282	11464
	2	9424	12635	10635	12812
	3	10537	12540	9880	12486
ProElastic	1	12154	14002	12645	12904
	2	12136	13661	12534	14382
	3	11841	13854	12530	14377

Both energy and time values were used to compute the results presented in Table 5. The results of cost emphasise the benefits of using cloud elasticity, where elastic execution obtained either a better or a competitive cost when compared with non-elastic executions. The best results of ProElastic appear when using 1 or 2 nodes: these situations present the advantages of the resource management on allocating and deallocating resources, which increases the application time but not causing a prohibitive use of energy for that. In addition, Table 5 presents the advantages of using ProElastic when confronted to AutoElastic: the former presents better execution time in all of the 12 cases (see Table 3) and a better cost in 7 of them (see Table 5).

Table 5 Cost in accordance with equation (3)

		Workload			
	Nodes	*Constant*	*Wave*	*Ascending*	*Decreasing*
Non-Elastic	1	32,691,698	33,093,180	32,417,352	33,280,561
	2	24,641,296	26,504,478	25,222,995	25,626,375
	3	18,083,912	20,355,942	18,691,350	21,823,708
	4	20,730,360	23,150,610	23,285,194	21,801,455
AutoElastic	1	22,404,204	28,512,900	22,404,478	23,432,416
	2	22,202,944	23,286,305	22,567,470	22,190,384
	3	18,502,972	22,170,720	20,668,960	20,177,376
ProElastic	1	22,132,434	28,480,068	21,155,085	24,259,520
	2	19,745,272	23,934,072	19,139,418	24,089,850
	3	18,353,550	22,997,640	19,095,720	21,795,532

6.2 Analysing the behaviour of the ProElastic execution

The idea of this subsection is to present graphs regarding the accuracy of the prediction and how resource reconfiguration happens along the time. Figures 9 , 10 and 11 depict the CPU load behaviour for the constant, ascending and descending workloads. For each graph in these figures, we present the real and the predicted CPU loads, the moments of scaling out operations, the moments where a new resource is delivered to the application and the warm-up prediction period. Particularly, this last information refers to the time that we gather CPU values but there are not enough quantitatively to start the ARIMA prediction engine. In Figure 9, we observed that the predicted values are very close to real ones. Two VMs were allocated when passing 75 s of execution, which were delivered after crossing 240 s. The same was perceived for 330 s and 495s. After adding 4 VMs, the application executes in steady way with a mean CPU load of 68%.

Figure 9 Application behaviour of the constant workload with ProElastic

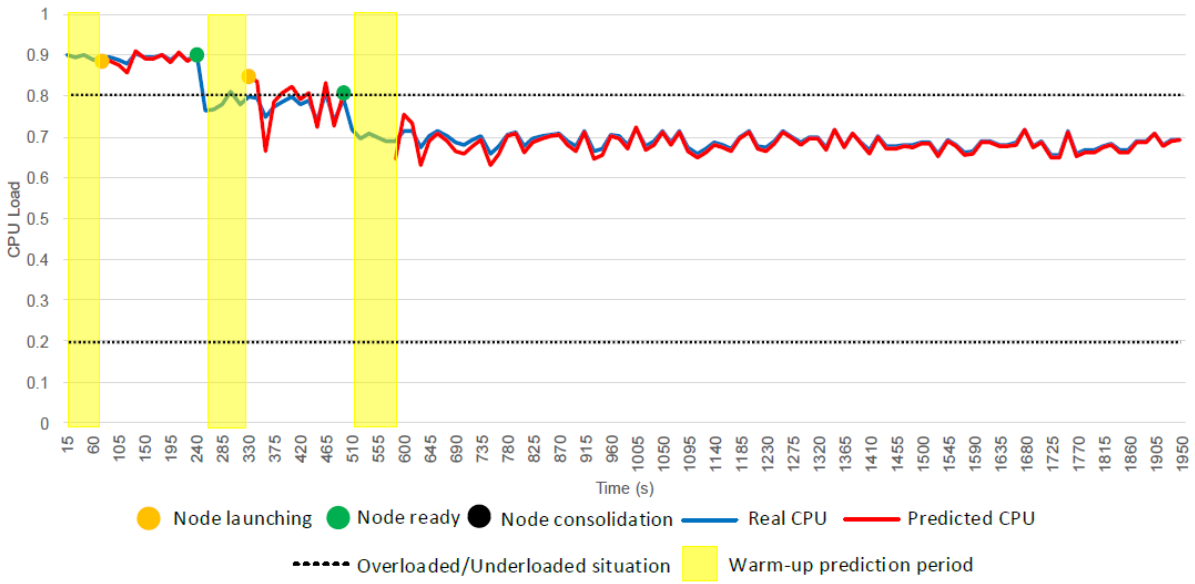


Figure 10 Application behaviour of the ascending workload with ProElastic

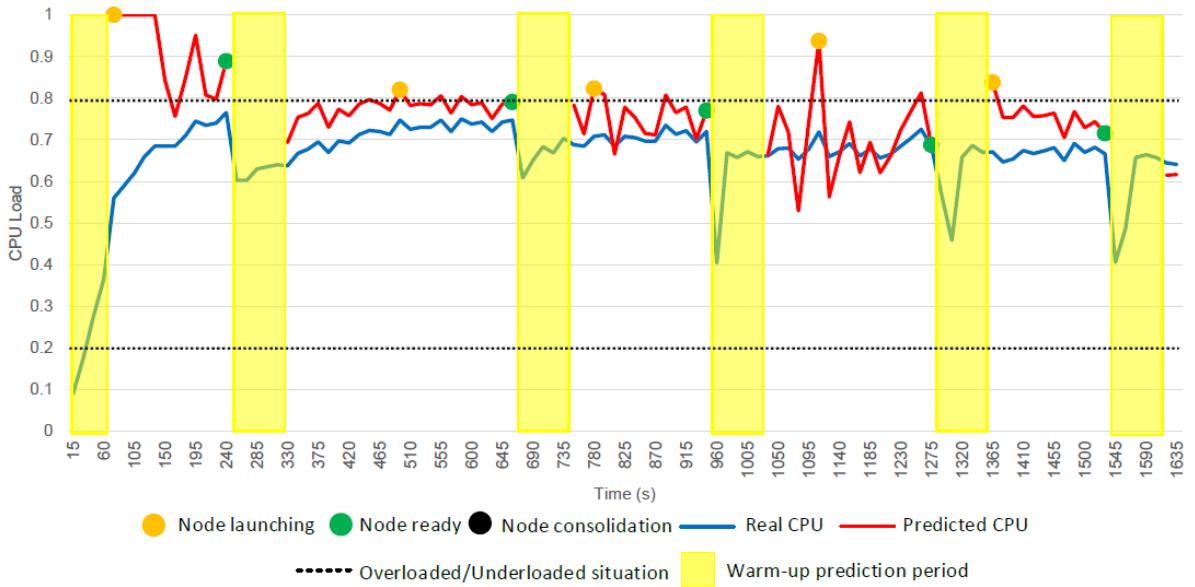


Figure 11 Application behaviour of the descending workload with ProElastic

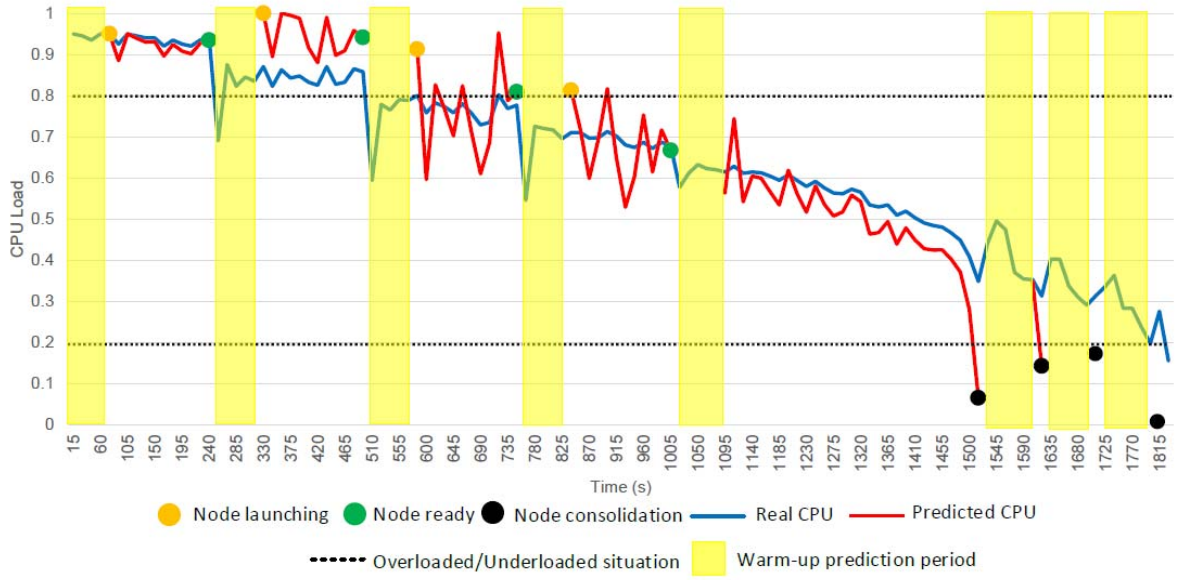


Figure 12 Resource allocation versus CPU load when executing the ascending workload. (a), (b) and (c) using ProElastic; (d), (e) and (f) using AutoElastic and (g), (h) and (i) using a non-elastic approach. We are starting with 1 node in the first row, 2 nodes in the second row and 3 nodes in the third row

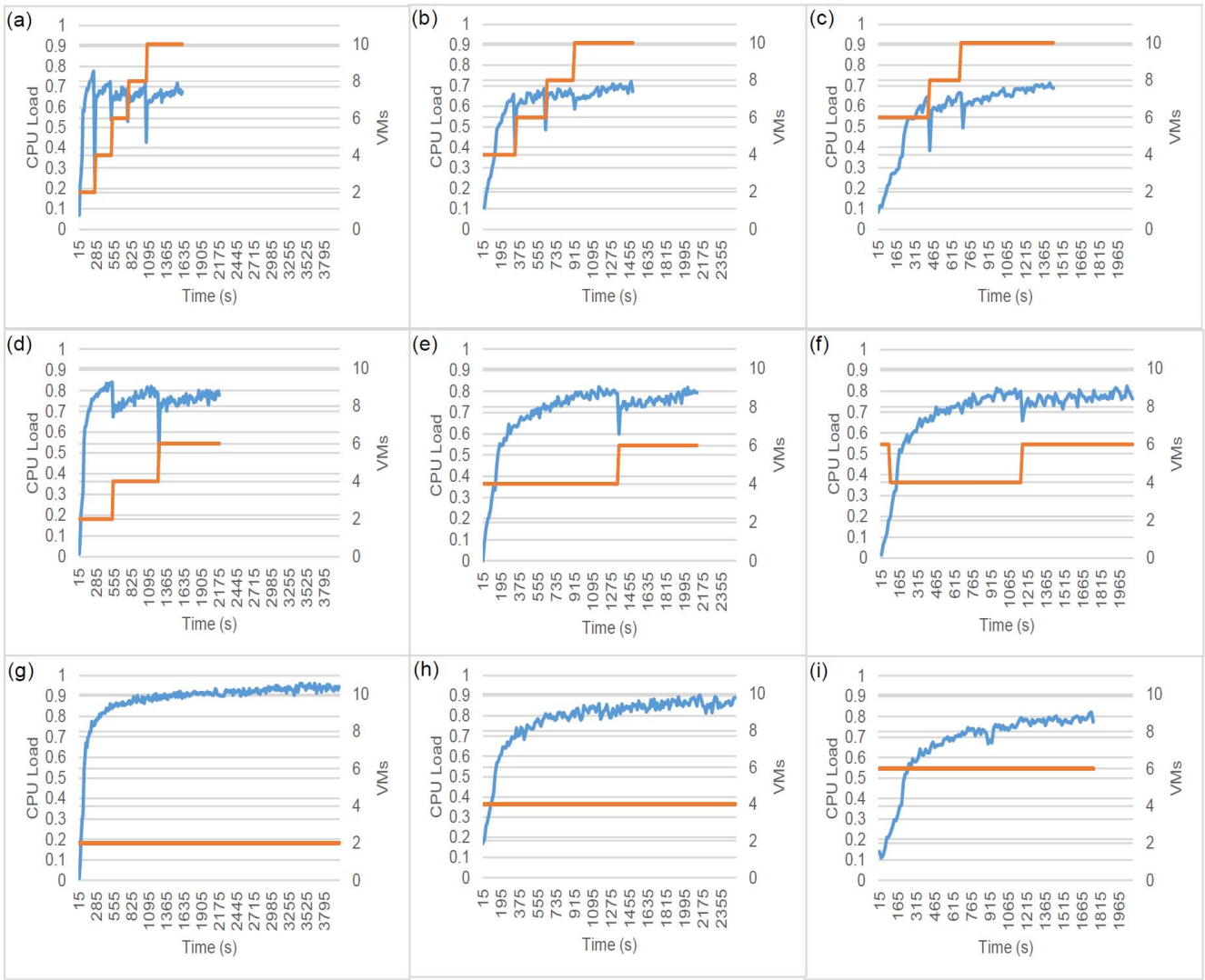
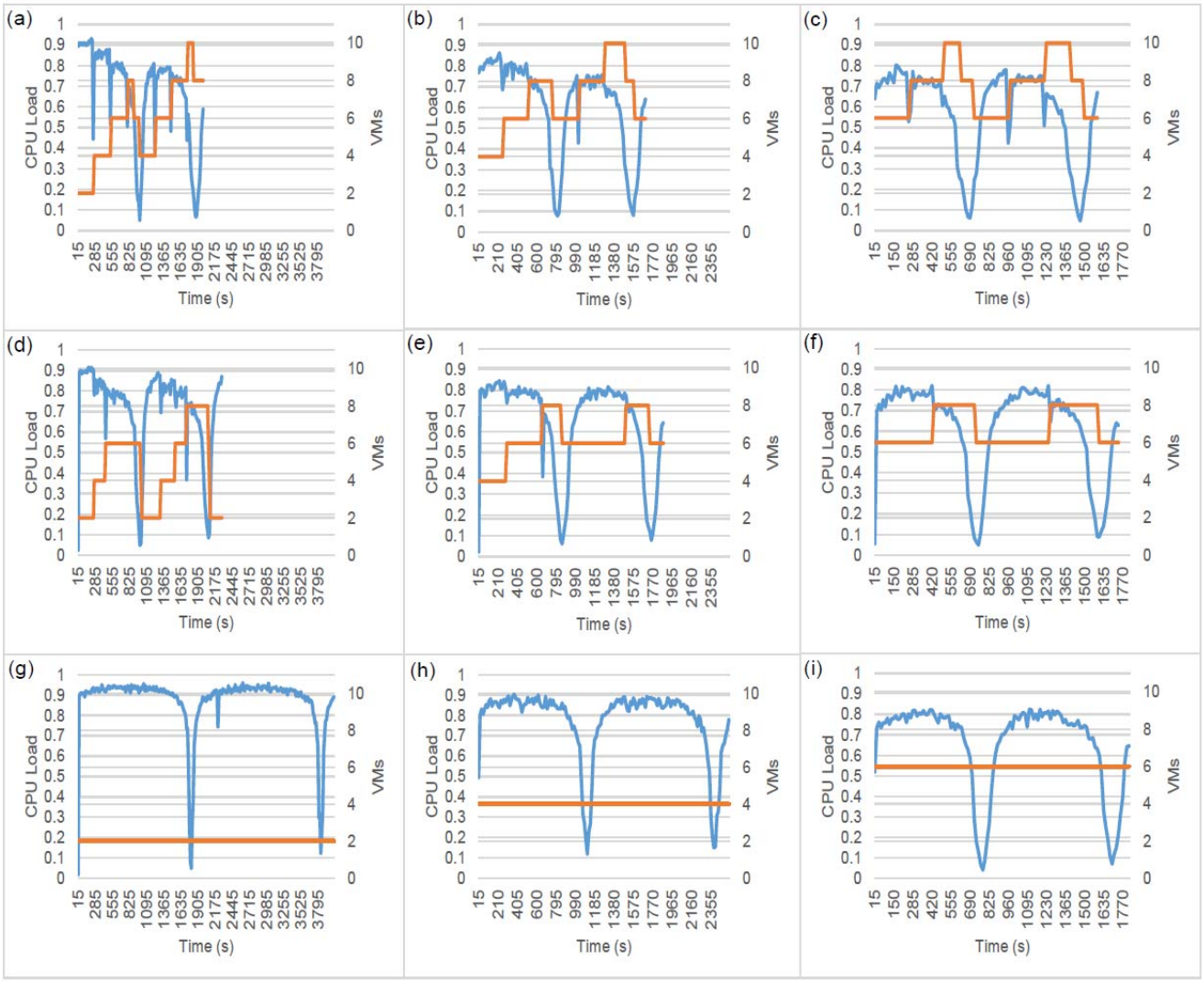


Figure 13 Resource allocation versus CPU load when executing the wave workload. (a), (b) and (c) using ProElastic; (d), (e) and (f) using AutoElastic and (g), (h) and (i) using a non-elastic approach. We are starting with 1 in the first row, 2 nodes in the second row and 3 nodes in the third row



Considering the ascending graph in Figure 10, the CPU load never reaches the upper threshold, i.e., we do not have violations. However, elasticity actions are not conducted by the CPU load itself, but by the predicted values; so, we have 5 moments of VM allocation which explains the obtained performance. In other words, the prediction was decisive for improving application performance by triggering scaling out operations accurately. The weakness point here is the wide variation between CPU load and the predicted value, where some peaks take place for this last metric. Considering the behaviour of the descending workload, Figure 11 also presents peaks in the prediction data. But these peaks were responsible, first for allocating resources to execute the application faster; second to deallocate resources since a descending pattern was detected in the time series.

Figures 12 and 13 illustrate the application behaviour of the three scenarios when executing the ascending and wave workloads. In parts (d), (e) and (f) of these figures, we can observe that resource allocation only happens when exceeding the thresholds and, consequently, the application executes in an overloaded state during a particular time up to delivering new VMs. Figures 12 and 13 also present three important information: (i) resource allocations are faster in ProElastic when confronted to AutoElastic, highlighting the lack of reactiveness of this last approach; (ii) an application compiled with ProElastic executes quicker if compared to the other scenarios (in some cases, more than 3 times faster); (iii) the allocation of more resources, but not stressing them, was responsible for the ProElastic results. Exploring information (iii), in terms of performance, it is better to have 2 nodes, each one with 70% of CPU load, instead of using only one node with 90% or more of CPU load. In addition, the results in Table 5 show that ProElastic's elasticity approach is not prohibitive; in contrary, we not only efficiently manage the number of resources to get performance but also obtained competitive or better values of cost.

In addition, Figure 12 presents a graph of the ascending workload that presents a relationship between CPU load and the number of running VMs. In this kind of graphs, it is pertinent to observe that exactly in the moment where a new resource is delivered, we have a better load balancing which involves the division of computing demands among a larger number of slave processes. Thus, at this moment, the load decreases in response to this resource reorganisation action. Immediately after providing a new resource, the load abruptly drops, so arising afterwards. In this figure, we observe the proactive nature of the ProElastic manager, which anticipates scaling out decisions so delivering resources always before reaching an index load indicated by the upper threshold.

7 Conclusion

Demand for HPC continues to grow, driven in large part by ever increasing demands for more accurate and faster simulations to meet new regulatory requirements, to increase safety or to reduce financial risks. Aiming at fitting this statement, this article presented a model named ProElastic and its functioning in the HPC scope to optimise cloud resource allocation in a proactive way. ProElastic acts at middleware level targeting iterative message-passing applications that can be easily implemented in MPI 2, which offers a sockets-based programming style for dynamic process creation. The use of time series and ARIMA-based prediction, together with an estimation of the scaling out operation time and the frequency of periodical monitoring, was decisive to accurately anticipate resource reconfiguration actions in such a way their delivery happens before the moment in which they are really needed by the application. The main scientific contribution is the ProElastic framework, which not only transforms a non-elastic application in an elastic one but also presents a communication architecture between application's processes and the elasticity manager. Moreover, considering the time requirements of HPC applications, we modelled a framework to enable a novel feature denoted asynchronous elasticity, where VM transferring or consolidation happens in parallel with application execution.

ProElastic was evaluated with a prototype that ran a numerical integration application when considering three metrics (time, energy and cost) and three scenarios (non-elastic, elastic with a reactive manager named AutoElastic and ProElastic). The results emphasised the performance gains with ProElastic when confronted to the other scenarios. These gains range from 7% to 48% in favor of ProElastic when analysing Table 3. Also pertinent, this reduction in the application time is accompanied with a non-prohibitive use of resources, as stated in the cost values obtained with ProElastic (see Table 5 for details).

Although achieving accuracy and performance, future research concerns the employment of other prediction policies, such as neural networks and SVM. In addition, the study of the elasticity grain and the execution of highly irregular applications also contemplate future steps. The grain, in particular, refers to the number of nodes and VMs involved on each elasticity action. Regarding the target application, although the numerical integration application be useful to evaluate ProElastic ideas, we intend to explore elasticity on highly-dynamic applications (Jin et al., 2014). Finally, our plans also consider to extend ProElastic to cover elasticity on other HPC programming models, such as divide-and-conquer, pipeline and bulk-synchronous parallel (BSP).

Acknowledgement

This work was partially supported by the following Brazilian agencies: CNPq, FAPERGS and CAPES.

References

Albonico, M., Mottu, J.-M. and Sunyé, G. (2016) 'Controlling the elasticity of web applications on cloud computing', *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC'16*, New York, NY, USA, ACM, pp.816–819.

Barrett, E., Howley, E. and Duggan, J. (2013) 'Applying reinforcement learning towards automating resource allocation and application scalability in the cloud', *Concurrency and Computation: Practice and Experience*, Vol. 25, No. 12, pp.1656–1674.

Beernaert, L., Matos, M., Vilaça, R. and Oliveira, R. (2012) 'Automatic elasticity in openstack', *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management, SDMCMM'12*, New York, NY, USA, ACM, pp.2:1–2:6.

Cai, B., Xu, F., Ye, F. and Zhou. W. (2012) 'Research and application of migrating legacy systems to the private cloud platform with cloudstack', *Automation and Logistics (ICAL), 2012 IEEE International Conference on*, pp.400–404.

Chang, F. (2016) 'Forecasting production quantity by integrating time series forecast technologies and artificial intelligence methods', *Proceedings of the 3rd Multidisciplinary International Social Networks Conference on Social Informatics 2016, Data Science 2016, MISNC, SI, DS 2016*, New York, NY, USA, ACM, pp.9:1–9:5.

Chilipirea, C., Constantin, A., Popa, D., Crintea, O. and Dobre, C. (2016) 'Cloud elasticity: going beyond demand as user load', *Proceedings of the 3rd International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, ARMS-CC'16*, New York, NY, USA, ACM, pp.46–51.

Coutinho, E.F., Rego, P.A.L., Gomes, D.G. and de Souza, J.N. (2016) 'An architecture for providing elasticity based on autonomic computing concepts', *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC'16*, New York, NY, USA, ACM, pp.412–419.

da Rosa Righi, R., Rodrigues, V.F., da Costa, C.A., Kreutz, D. and Heiss, H.-U. (2015) 'Towards cloud-based asynchronous elasticity for iterative HPC applications', *Journal of Physics: Conference Series*, Vol. 649, No. 1, 012006.

El Hag, H. and Sharif, S.M. (2007) 'An adjusted ARIMA model for internet traffic', *AFRICON 2007*, pp.1–6.

Galante, G., De Bona, L.C.E., Mury, A.R., Schulze, B. and da Rosa Righi, R. (2016) 'An analysis of public clouds elasticity in the execution of scientific applications: a survey', *Journal of Grid Computing*, Vol. 14, No. 2, pp.193–216.

Gong, Z., Gu, X. and Wilkes, J. (2010) 'Press: predictive elastic resource scaling for cloud systems', *Network and Service Management (CNSM), 2010 International Conference on*, Niagara Falls, ON, IEEE, pp.9–16.

Harvey, P., Bakanov, K., Spence, I. and Nikolopoulos, D.S. (2016) 'A scalable runtime for the ecoscale heterogeneous exascale hardware platform', *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS'16*, New York, NY, USA, ACM, pp.7:1–7:8.

Islam, S., Lee, K., Fekete, A. and Liu, A. (2012) 'How a consumer can measure elasticity for cloud platforms', *Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12*, New York, NY, USA, ACM, pp.85–96.

Jin, L., Cong, D., Guangyi, L. and Jilai, Y. (2014) 'Short-term net feeder load forecasting of microgrid considering weather conditions', *Energy Conference (ENERGYCON), 2014 IEEE International*, May, pp.1205–1209.

Kalpakis, K., Gada, D. and Puttagunta, V. (2001) 'Distance measures for effective clustering of ARIMA time-series', *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, IEEE, pp.273–280.

Loff, J. and Garcia, J. (2014) 'Vadara: predictive elasticity for cloud applications', *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, Singapore, IEEE, pp.541–546.

Lorido-Botrán, T., Miguel-Alonso, J. and Lozano, J.A. (2012) *Auto-scaling Techniques for Elastic Applications in Cloud Environments*, Research EHU-KAT-IK, Department of Computer Architecture and Technology, UPV/EHU.

Lynar, T.M., Herbert, R.D. and Chivers, W.J. (2013) 'Reducing energy consumption in distributed computing through economic resource allocation', *International Journal of Grid and Utility Computing*, Vol. 4, No. 4, pp.231–241.

Lynar, T.M., Herbert, R.D., Chivers, S. and Chivers. W.J. (2011) 'Resource allocation to conserve energy in distributed computing', *International Journal of Grid and Utility Computing*, Vol. 2, No. 1, pp.1–10.

Martineau, M., McIntosh-Smith, S., Boulton, M. and Gaudin, W. (2016) 'An evaluation of emerging many-core parallel programming models', *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'16*, New York, NY, USA, ACM, pp.1–10.

Masood, W. and Schmidt, J.F. (2015) 'Exploring autoregressive integrated models for time synchronization in wireless sensor networks', *Proceedings of the 2015 Workshop on Wireless of the Students, by the Students, & for the Students, S3'15*, New York, NY, USA, ACM, pp.31–33.

Moore, L.R., Bean, K. and Ellahi, T. (2013) 'Transforming reactive auto-scaling into proactive auto-scaling', *Proceedings of the 3rd International Workshop on Cloud Data and Platforms, CloudDP'13*, New York, NY, USA, ACM, pp.7–12.

Moreno-Vozmediano, R., Montero, R.S. and Llorente, I.M. (2012) 'Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures', *Computer*, Vol. 45, No. 12, pp.65–72.

Netto, M.A.S., Cardonha, C., Cunha, R.L.F. and Assuncao, M.D. (2014) 'Evaluating auto-scaling strategies for cloud computing environments', *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2014*, Paris, France, 9–11 September, pp.187–196.

Nikravesh, A.Y., Ajila, S.A. and Lung, C.-H. (2015) 'Towards an autonomic auto-scaling prediction system for cloud resource provisioning', *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, Florence, IEEE, pp.35–45.

Perez-Palacin, D., Mirandola, R. and Scoppetta, M. (2016) 'Simulation of techniques to improve the utilization of cloud elasticity in workload-aware adaptive software', *Companion Publication for ACM/SPEC on International Conference on Performance Engineering, ICPE'16 Companion*, New York, NY, USA, ACM, pp.51–56.

Raveendran, A., Bicer, T. and Agrawal, G. (2011) 'A framework for elastic execution of existing mpi programs', *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW'11*, Washington, DC, USA, IEEE Computer Society, pp.940–947.

Rosa, B.A., Frederico, V.A., Bittencourt, L.F., Pereira, M.B. and Hisatomi, K.S. (2014) 'An experimental tool for elasticity management through prediction mechanisms', *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC'14*, Washington, DC, USA, IEEE Computer Society, pp.511–516.

Roy, N., Dubey, A. and Gokhale, A. (2011) 'Efficient autoscaling in the cloud using predictive models for workload forecasting', *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, Washington, DC, IEEE, pp.500–507.

Weidner, O., Atkinson, M., Barker, A. and Vicente, R.F. (2016) 'Rethinking high performance computing platforms: Challenges, opportunities and recommendations', *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing, DIDC'16*, New York, NY, USA, ACM, pp.19–26.

Zomaya, A.Y. (1996) *Parallel and Distributed Computing Handbook*, Vol. 718, McGraw-Hill, New York, USA.