SelfElastic: Combinando Elasticidade Reativa e Proativa para Aplicações de Alto Desempenho

Vinicius F. Rodrigues[1], Gustavo Rostirolla[1]
Rodrigo R. Righi[1], Cristiano A. Costa[1]

[1]Programa Interdisciplinar em Computação Aplicada (PIPCA) - Unisinos

viniciusfacco@live.com, grostirolla1@gmail.com,{rrrighi,cac}@unisinos.br

Abstract. *Cloud elasticity can bring benefits to High Performance Computing, providing a better use of resources and also reducing the execution time of applications. The most common approaches use threshold-based reactive elasticity or time-consuming proactive elasticity. However, both present at least one problem related to the need of a previous user experience, completion of parameters or design for a specific infrastructure and workload setting. In this context, we developed a self-organizing hybrid elasticity model for parallel applications named SelfElastic. The proposal presents a closed control loop elasticity architecture that adapts at runtime the thresholds' values. Based on SelfElastic, we developed a prototype in which the results were promising in terms of application execution time and cost when compared to a non-elastic execution.*

Resumo. *A elasticidade em nuvem pode trazer benefícios para a área de Computação de Alto Desempenho, como uma melhor utilização de recursos e redução do tempo de execução de aplicações. As abordagens mais comuns utilizam elasticidade reativa baseada em thresholds ou elasticidade proativa, a qual pode ser muito custosa computacionalmente. Ambas apresentam pelo menos um problema relacionado à necessidade de experiência prévia do usuário, parametrizações prévias ou modelagem para configurações de cargas de trabalho e infraestruturas específicas. Neste contexto, este trabalho apresenta SelfElastic - um modelo híbrido de elasticidade cuja a arquitetura possui um ciclo fechado de controle que adapta em tempo de execução os valores dos thresholds. Baseado em SelfElastic, foi construído um protótipo que apresentou resultados promissores em termos de tempo de execução e custo quando comparado à execuções não elásticas.*

1. Introdução

A adoção da Computação em Nuvem no cenário da Computação de Alto Desempenho vem aumentando, principalmente devido à sua característica conhecida como elasticidade [Herbst et al. 2015]. Esta característica permite o dimensionamento dos recursos de acordo com a demanda da aplicação. Entretanto, decidir a quantidade correta de recursos para aplicações paralelas em um ambiente de computação em nuvem torna-se um desafio, pois um dimensionamento *aplicação X recursos* inadequado pode resultar em situações conhecidas como *overprovisioning* ou *underprovisioning* [Nikravesh et al. 2015, Dustdar et al. 2015].

A maioria das estratégias de controle de elasticidade podem ser classificadas em reativas ou proativas/preditivas [Farokhi et al. 2015, Nikravesh et al. 2015]. Para o primeiro caso, tipicamente usuários definem *thresholds*, um superior e outro inferior, para uma determinada métrica de desempenho, ativando o aumento ou a diminuição de uma determinada quantidade de recursos ao ambiente [Netto et al. 2014]. Dependendo do *threshold*, tal técnica pode implicar em falta de reatividade, ou seja, o comportamento da aplicação remete a uma necessidade de elasticidade, mas ela não ocorre de imediato porque os *thresholds* ainda não foram atingidos. Por outro lado, uma abordagem proativa aplica técnicas de predição para antecipar o comportamento (carga) do sistema e assim decidir as ações de reconfiguração. Essa capacidade permite à aplicação estar pronta para tratar o aumento de carga quando ela ocorrer. Para esta estratégia, é comum o uso de algoritmos de aprendizagem de máquina incluindo Redes Neurais, Regressão Linear e técnicas de reconhecimento de padrões [Farokhi et al. 2015].

Apesar de a palavra automático ser utilizada em ambos os mecanismos de elasticidade, as implementações atuais normalmente requerem algum tipo de entrada do usuário, configurações preliminares e/ou utilização de APIs (*Application Programming Interface*) para ajustar os recursos conforme a carga de trabalho [Herbst et al. 2015]. Estas tarefas não são triviais e em alguns casos é necessário um profundo conhecimento do comportamento da aplicação sob um determinado hardware [Dustdar et al. 2015]. Nas estratégias reativas, isso torna a acurácia do conjunto de *thresholds* incerta, pois o mesmo conjunto que serve bem para uma determinada configuração de infraestrutura e aplicação pode obter comportamentos indesejados em outras [Netto et al. 2014]. Já estratégias proativas, apesar de não necessitarem de *thresholds*, são baseadas em modelos matemáticos robustos e normalmente classificadas como consumidores de tempo para aplicações de alto desempenho [Nikravesh et al. 2015]. Além disso, normalmente ocorre também a necessidade de treinamento da técnica preditiva e prévia execução da aplicação para otimizar a seleção de parâmetros [Farokhi et al. 2015].

Neste contexto, foi proposto em trabalho prévio o modelo chamado AutoElastic [Righi et al. 2016] o qual foca na elasticidade reativa para aplicações paralelas. Apesar dos ganhos de desempenho obtidos, AutoElastic ainda sofre dos principais problemas de abordagens reativas: definição de *thresholds* e falta de reatividade. Considerando isso, o presente artigo propõe um modelo de elasticidade chamado SelfElastic, que adota uma abordagem de elasticidade híbrida, combinando características de reatividade e proatividade. SelfElastic age como um serviço de provisionamento de recursos em nível PaaS (*Platform as a Service*) de uma nuvem, unindo a característica baseada em *thresholds* da abordagem reativa ao controle de realimentação e predição da abordagem proativa. Para tal, SelfElastic fornece um *framework* com um controlador que transparentemente gerencia operações de elasticidade horizontal e adapta os valores dos *thresholds* para que se obtenha melhor reatividade, sem a necessidade de modificações ou adaptações na aplicação e sem intervenções do usuário. Além do modelo SelfElastic, esse artigo também apresenta um protótipo e sua avaliação, cujos resultados são promissores em relação a tempo de execução da aplicação paralela e custo (tempo $\times$ uso de recursos) quando comparados àqueles obtidos em uma execução não elástica.

2. Trabalhos Relacionados

Esta seção apresenta os trabalhos relacionados a elasticidade em nuvem, contemplando tanto iniciativas privadas (comerciais), quanto iniciativas acadêmicas. Iniciativas comercias normalmente oferecem a elasticidade de forma manual, considerando a percepção do usuário, ou através de configurações prévias (elasticidade reativa). Na técnica manual, usuários podem desenvolver suas próprias aplicações para monitoramento de dados e serviços que executam em máquinas virtuais na nuvem, gerenciando as operações de elasticidade quando necessário. Porém, algumas plataformas como Amazon AWS (https://aws.amazon.com/pt/), Microsoft Azure (https://azure.microsoft.com/pt-br/) e Nimbus (http://www.nimbusproject.org/) disponibilizam sistemas configuráveis para monitoramento de serviços e gerenciamento de elasticidade [Chiu and Agrawal 2010, Roloff et al. 2012]. Neste caso, os usuários devem definir regras e *thresholds* sob determinadas métricas, as quais são monitoradas e assim que condições sejam satisfeitas ações de elasticidade são executadas. Em particular, tanto a Amazon AWS [Chiu and Agrawal 2010] quanto a Microsoft Azure [Roloff et al. 2012] esperam uma quantidade específica de observações de carga consecutivas fora da margem de um *threshold* para executar ações de elasticidade.

Quanto as iniciativas de pesquisa acadêmica, o estudo da literatura apontou alguns pontos fracos em relação ao tratamento da elasticidade, os quais são explorados abaixo:

(i) Nenhuma análise de situações de pico esporádicos (ação de elasticidade conhecida como falso-positiva) quando se atinge um *threshold* [Beernaert et al. 2012];
(ii) Necessidade de alterar o código fonte da aplicação ou desenvolver *scripts* adicionais [Raveendran et al. 2011, Rajan et al. 2011, Mariani et al. 2014, Copil et al. 2013];
(iii) Utilização de componentes proprietários que não estão disponíveis como bibliotecas de programação para sistemas operacionais conhecidos, tais como GNU-Linux, Windows e MacOS [Raveendran et al. 2011, Beernaert et al. 2012];
(iv) Necessidade de conhecer o comportamento da aplicação com antecedência, como o tempo de execução esperado dos componentes [Raveendran et al. 2011];
(v) Reconfiguração de recursos com a abordagem *stop-reconfigure-and-go* [Raveendran et al. 2011].

Considerando a área específica de aplicações paralelas e elasticidade, destacam-se algumas iniciativas: ElasticMPI [Raveendran et al. 2011], Work Queue [Rajan et al. 2011], Jackson et al. [Jackson et al. 2010], Mariani et al. [Mariani et al. 2014] e Spinner et al. [Spinner et al. 2014]. Entre elas, três iniciativas executam aplicações iterativas, onde cada nova fase significa que há um novo esforço para redistribuir as tarefas aos escravos [Raveendran et al. 2011, Rajan et al. 2011, Jackson et al. 2010]. A elasticidade em Rajan et al. [Rajan et al. 2011] é oferecida manualmente, onde o usuário captura dados de monitoramento, utilizando a estrutura proposta pelos autores, e adiciona ou remove recursos do ambiente. Jackson et al. [Jackson et al. 2010] executam aplicações mestre-escravo da NERSC como *benchmark* para medir o desempenho de

configurações de um *cluster* no Amazon EC2. Já Spinner et al. [Spinner et al. 2014] propõem um *middleware* que fornece elasticidade vertical para máquinas virtuais que executam aplicações HPC. Eles argumentam que a abordagem horizontal é proibitiva no âmbito HPC porque, segundo eles, a inicialização de uma nova máquina virtual leva pelo menos 1 minuto. Por fim, a elasticidade no trabalho de Mariani et al. [Mariani et al. 2014] é oferecida em nível de API, na qual o usuário gerencia a reconfigurações de recursos. Esta estratégia requer experiência por parte do programador e não seria portável entre diferentes ambientes de nuvem. Levando em conta a análise realizada, percebe-se uma lacuna para exploração da elasticidade que combina as abordagens proativa e reativa e que não necessite conhecimento prévio do comportamento da aplicação, nem alteração do código das aplicações paralelas.

3. Modelo SelfElastic

SelfElastic é um modelo que fornece elasticidade horizontal e híbrida de forma transparente para aplicações paralelas. O termo transparente diz respeito ao fato de não necessitar a intervenção do usuário para definição de regras e ações, também não necessitando modificações no código fonte da aplicação paralela. O termo híbrido se refere a uma combinação de elasticidade reativa e proativa, neste caso, optou-se pela utilização de *thresholds* mas estes agora são maleáveis e dirigidos por algoritmos que visam aumentar o desempenho da aplicação. Utilizou-se a elasticidade horizontal, pois no caso da vertical a quantidade de recursos fica limitada aos disponíveis em uma única máquina física e, além disso, a maioria dos sistemas operacionais não permitem que sejam adicionados recursos sem a necessidade de reinicialização [Lorido-Botran et al. 2014].

A Figura 1 apresenta a arquitetura do modelo, bem como o mapeamento de máquinas físicas e processos em um ambiente de nuvem. O *framework* inclui um componente de gerenciamento do sistema de elasticidade composto por três módulos. O Gerenciador de Elasticidade não possui dependência de localização para sua execução (podendo estar dentro ou fora da nuvem), necessitando apenas de conexão com o Front-End da nuvem. Isso é possível através do uso da API fornecida pela plataforma de nuvem. A arquitetura também contempla uma área de dados compartilhada entre todos os componentes do ambiente, incluindo o Gerenciador de Elasticidade, que acessa esta área através do Front-End da nuvem. A aplicação do usuário, ao ser executada no ambiente, também possui acesso a área de dados através da máquina virtual na qual ela está sendo executada. Isso possibilita a implementação de políticas de comunicação entre os componentes e processos do ambiente. Mais precisamente, o Gerenciador escreve nesta área quando ações de elasticidade acontecem, enquanto que os processos da plicação realizam a leituras para verificar se novas reconfigurações de recursos e processos são necessárias.

O sistema de elasticidade considera dados de carga de trabalho das máquinas virtuais que compõem a infraestrutura de nuvem como entrada para análise de necessidades de reconfigurações de recursos para aplicações paralelas iterativas que seguem o modelo mestre-escravo. Embora trivial, este tipo de construção é utilizado em muitas áreas, tais como algoritmos genéticos, técnicas de Monte Carlo, transformações geométricas em computação gráfica e algoritmos de criptografia [Raveendran et al. 2011]. Quando uma aplicação é compilada, SelfElastic se

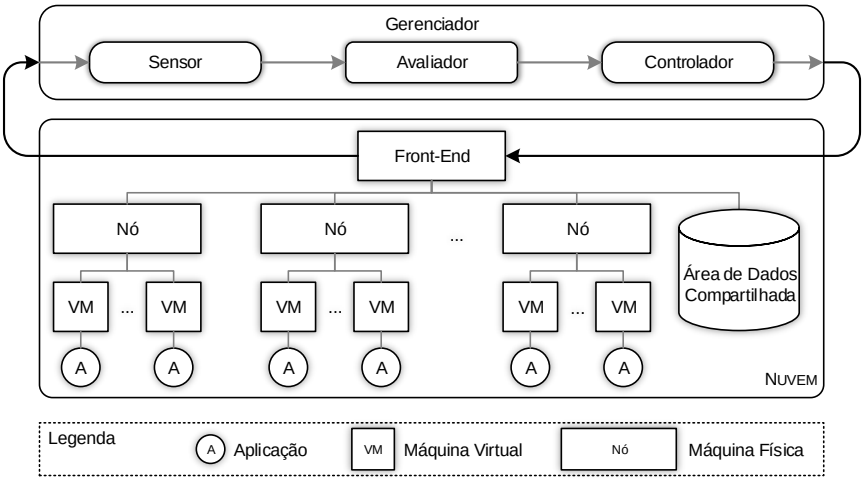


Figura 1. Arquitetura do modelo SelfElastic.

encarrega de adicionar o código de controle de elasticidade no início de cada *loop* do processo mestre da aplicação. Tal código faz com que o mestre cheque a área de dados compartilhada e redimensione os processos necessário para a computação daquela iteração.

A atividade de monitoramento do sistema de elasticidade é realizada periodicamente, assim como é feita em outras iniciativas [Chiu and Agrawal 2010, Imai et al. 2012]. Considerando que as aplicações de alto desempenho são conhecidas como sendo de computação intensiva, SelfElastic utiliza no momento apenas a métrica CPU coletada de cada máquina virtual para monitoramento e definição dos thresholds. Dessa maneira, o módulo Sensor captura periodicamente valores de carga de CPU de cada máquina virtual que executa processos escravos, enquanto o Avaliador aplica um cálculo de séries temporais sobre estes valores, considerando também coletas anteriores de dados. Assim, esses dados são utilizados para calcular a carga do sistema (CPS) a qual é utilizada pelo módulo Avaliador para a adaptação dos thresholds e pelo módulo Controlador para a tomada de decisões.

Ao invés da utilização de *thresholds* fixos previamente parametrizados para a tomada de decisões de elasticidade, o módulo Avaliador realiza a adaptação dinâmica dos *thresholds* inferior e superior, os quais são inicializados com os valores 0% e 100%, respectivamente. O módulo Avaliador possui dois procedimentos que formam a técnica nomeada de Live thresholding: adapta_thresholds() e recalcula_thresholds(). O primeiro é computado a cada observação de monitoramento, já o segundo é chamado apenas quando uma ação de elasticidade é realizada pelo módulo Controlador. Ambos foram modelados para aumentar a reatividade da elasticidade do ambiente, buscando oferecer valores competitivos de tempo de execução da aplicação ($tempo$) e custo ($tempo \times recursos$) quando comparados com os cenários da tradicional técnica de *thresholds* fixos e a não utilização de elasticidade.

3.1. adapta_thresholds(): Redefinindo os *thresholds* a cada Observação de Monitoramento

Este procedimento possui três parâmetros: T_i, T_s (representando respectivamente o threshold inferior e superior) e $carga$. Primeiramente, é computada a variação da carga do sistema considerando a carga atual ($CPS(o)$) e a carga da observação

anterior ($CPS(o-1)$), atribuindo esse valor para ΔCPS (Equação 1). ΔCPS define qual *threshold* será atualizado: (i) se ΔCPS é negativo, a carga do sistema está diminuindo, então T_i é recalculado para lidar com essa situação rapidamente; (ii) se ΔCPS é positivo, a carga do sistema está aumentando, então T_s é atualizado para resolver esta situação; (iii) se ΔCPS é igual a 0, adaptações de *thresholds* não ocorrem devido à carga do sistema ter se mantido estável. As Equações 2 e 3 demonstram como os novos valores dos *thresholds* são computados, em que Min e Max retornam respectivamente o menor e maior valor passado por parâmetro. Contemplando que T_s diminui quando atualizado, esse *threshold* possui um limite inferior igual a 0. Da mesma maneira, um limite superior de 100 é utilizado ao computar o novo valor de T_i, o qual aumenta quando atualizado.

$$\Delta CPS = CPS(o) - CPS(o-1) \tag{1}$$

$$T_i = Min(T_l + |\Delta CPS|, 100) \tag{2}$$

$$T_s = Max(T_u - \Delta CPS, 0) \tag{3}$$

3.2. recalcula_thresholds(): Redefinindo os thresholds Quando um Deles é Violado

Como estratégia inicial para definir o recalcula_thresholds(), o qual possui os mesmos parâmetros apresentados em adapta_thresholds(), uma possibilidade é reatribuir aos *thresholds* seus valores iniciais (0 e 100) a cada ação de elasticidade. Esta estratégia pode não ser a melhor para aumentar a reatividade da elasticidade, pois os dados históricos de operações realizadas são perdidos, não podendo ser utilizados para influenciar decisões futuras por parte do Gerenciador. Ou seja, caso T_s seja violado em 75%, ele voltará a 100%, mas a aplicação pode demorar um tempo significativo para que ele baixe novamente para um valor próximo de 75%, executando então com recursos saturados. Buscando propor novas formas de recalcular os *thresholds* quando um deles é violado, foi analisado o algoritmo de congestionamento do protocolo TCP [Tanenbaum 2003]. A Figura 2 ilustra um exemplo desse algoritmo, demonstrando também pontos de analogia com o conceito de elasticidade em nuvem baseada em *thresholds*. Dessa maneira, o é o índice da observação de monitoramento após uma ação de elasticidade (ponto 6) e, $CPS(o)$ e $CPS(o-1)$ a carga do sistema nas observações o e $o-1$ (ponto 5).

Assim, foram investigadas 6 abordagens diferentes A_z ($A_z|z \in \{a, b, c, d, e, f\}$) para tratar a adaptatividade dos *thresholds* após uma ação de elasticidade (ilustrado no ponto 7): ao violar T_i podem ser aplicados A_a, A_b ou A_c para computar o novo valor para T_i, enquanto T_s é redefinido para 100; ao violar T_s podem ser aplicados A_d, A_e ou A_f, enquanto T_i é reiniciado para 0. O Gerenciador SelfElastic sempre utiliza uma combinação fixa de uma abordagem quando violado T_i e outra para T_s. Esta combinação resulta em uma notação nomeada LT_{xy}, em que x (x é A_a, A_b ou A_c) e y (y é A_d, A_e ou A_f) se referem a uma possibilidade particular para os *thresholds* inferior e superior, respectivamente. Em resumo, recalcula_thresholds() depende de qual *threshold* foi violado: se T_i, seu valor é computado por uma das três abordagens da Equação 4 e T_s volta para 100; se T_s, seu valor é calculado por uma das três abordagens da Equação 5 e T_i volta para 0.

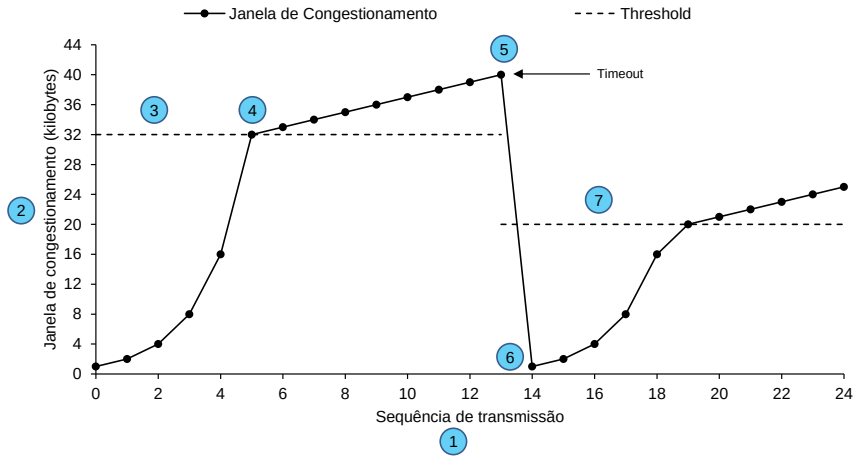


Figura 2. Exemplo do algoritmo de congestionamento TCP [Tanenbaum 2003].

$$T_i = \begin{cases} 0 & para\ A_a \\ \frac{CPS(o)}{2} & para\ A_b \\ CPS(o-1) - \left|\frac{CPS(o-1)-CPS(o)}{2}\right| & para\ A_c \end{cases} \quad (4)$$

$$T_s = \begin{cases} 1 & para\ A_d \\ CPS(o) + \frac{1-CPS(o)}{2} & para\ A_e \\ CPS(o-1) + \left|\frac{CPS(o-1)-CPS(o)}{2}\right| & para\ A_f \end{cases} \quad (5)$$

4. Metodologia de Avaliação

Para a realização dos experimentos, foram utilizados 11 nós computacionais homogêneos com processadores de dois núcleos de 2.9 GHz e 4 GB de memória, interconectados através de uma rede 100 Mbps. A plataforma de nuvem utilizada foi a OpenNebula versão 4.12.1. Um nó atua como Front-End e servidor para o OpenNebula, enquanto que os 10 restantes foram configurados para receberem máquinas virtuais. Para a avaliação do modelo SelfElastic, foi implementado um protótipo do Gerenciador SelfElastic utilizando Java. O controle e monitoramento do ambiente de nuvem é realizado através da API Java fornecida pelo próprio *middleware* OpenNebula. O Gerenciador SelfElastic possibilita que seja informado o SLA através de um arquivo XML que respeite o padrão WS-Agreement[1]. Esse arquivo é passado como parâmetro ao gerenciador contendo a quantidade máxima e mínima de recursos para executar a aplicação. Na execução elástica, parte-se de 1 nó (2 VMs, 1 por núcleo de processamento) podendo chegar até 10 nós (20 VMs). Já na não elástica, a execução é iniciada e finalizada com 2 VMs.

A aplicação utilizada nos testes calcula a aproximação para a integral do polinômio $f(x)$ num intervalo fechado $[a, b]$. Para tal, foi implementado o método

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

de Newton-Cotes conhecido como Regra do Trapézio Repetida [Comanescu 2012]. Considere a partição do intervalo $[a, b]$ em s subintervalos iguais, cada qual de comprimento h ($[x_i, x_{i+1}]$, para $i = 0, 1, 2, ..., s-1$). Assim, $x_{i+1} - x_i = h = \frac{b-a}{s}$. Dessa forma, pode-se escrever a integral de $f(x)$ como sendo a soma das áreas dos s trapézios contidos dentro do intervalo $[a, b]$ como apresentado na Equação 6. s representa a quantidade de subintervalos, sendo assim existem $s+1$ equações $f(x)$ para se obter o resultado final da integral. O processo mestre deve distribuir essas $s+1$ equações entre os processos escravos. Como s define a quantidade de subintervalos, e consequentemente a quantidade de equações para computar a integral, quanto maior for esse parâmetro, maior é a carga computacional para atingir o resultado final.

$$\int_a^b f(x)\,dx \approx A_0 + A_1 + A_2 + A_3 + ... + A_{s-1} \tag{6}$$

em que A_i = area do trapezoide i, com $i = 0, 1, 2, 3, ..., s-1$.

A carga de trabalho recebida pelo processo mestre consiste em uma lista de equações e seus parâmetros, incluindo o intervalo $[a, b]$ e quantidade de subintervalos s para cada uma, enquanto o retorno é um vetor com a integral de cada equação. Buscando analisar diferentes situações de elasticidade, s foi utilizado para modelar quatro comportamentos de carga de computação: Constante, Crescente, Decrescente e Onda. A cada iteração, uma equação é selecionada para cálculo e o parâmetro s é recalculado individualmente, modelando um determinado comportamento de carga. A Figura 3 apresenta graficamente uma representação de cada comportamento de carga. O eixo x expressa a iteração (ou seja, uma determinada equação que será distribuída aos escravos pelo processo mestre), enquanto que o eixo y representa a respectiva carga de processamento para aquela iteração (valor de s).

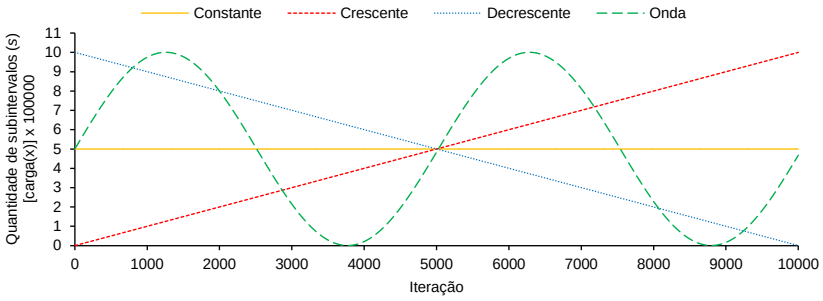


Figura 3. Visão gráfica dos comportamentos de carga. Cada iteração corresponde a uma equação a ser calculada.

5. Resultados

A Tabela 1 apresenta os resultados considerando as nove combinações possíveis das abordagens de Live Thresholding e os quatro comportamentos de carga desenvolvidos para a aplicação. A tabela também apresenta os resultados da execução sem elasticidade. Nessa tabela, *Recurso* é estimado através de $\sum_{i=0}^{z} VMs(i)$, onde z é o número total de observações de monitoramento e $VMs(i)$ é o número de VMs em execução na observação de ordem i. Ainda, *Custo* é obtido pela multiplicação de *Tempo* por *Recurso*. Através da comparação das execuções elásticas com a não elástica, é possível notar que as primeiras obtiveram índices de desempenho melhores

que chegaram a 54,9%, 59%, 54,6% e 50,4%, respectivamente para os comportamentos de carga Constante, Crescente, Decrescente e Onda. Isso se deve ao fato de que com a elasticidade, ao logo da execução mais recursos possibilitam que a carga da aplicação seja melhor distribuída acelerando o término da aplicação. Por outro lado, é importante destacar que para isso, mais recursos são necessários elevando o índice de consumo de recursos.

Tabela 1. Resultados incluindo todas as abordagens para adaptar T_i(A_a, A_b e A_c) e T_s(A_d, A_e e A_f) em comparação com a execução da aplicação sem elasticidade (SE). Os valores em *itálico* e negrito representam respectivamente o melhor e o pior resultado para cada métrica.

LT_{xy}		Constante			Crescente			Decrescente			Onda		
T_i	T_s	Tempo	Recurso	Custo	Tempo	Recurso	Custo	Tempo	Recurso	Custo	Tempo	Recurso	Custo
	A_d	2380	10836	25790	2084	10924	22766	**2397**	11100	26607	2334	11700	27308
A_a	A_e	2267	11366	25767	1987	10824	21507	2243	12288	27562	2376	11602	27566
	A_f	2266	11176	25325	1893	11804	22345	2251	12252	27579	2274	11724	*26660*
	A_d	2413	10866	26220	2130	10836	**23081**	2305	12160	28029	**2674**	10514	28114
A_b	A_e	2382	11058	26340	1931	11548	22299	2138	**13662**	**29209**	2221	12634	28060
	A_f	2338	11032	25793	1930	11384	21971	2271	12310	27956	2609	11036	28793
	A_d	**2502**	*10572*	**26451**	**2138**	*10590*	22641	2274	*10920*	*24832*	2660	10472	27856
A_c	A_e	*1932*	**12828**	*24784*	*1769*	**12064**	*21341*	*2000*	13088	26176	*2165*	**13408**	**29028**
	A_f	2292	11404	26138	1879	11686	21958	2245	12036	27021	2603	*10426*	27139
SE		4283	8542	36585	4319	8618	37221	4410	8798	38799	4363	8700	37958

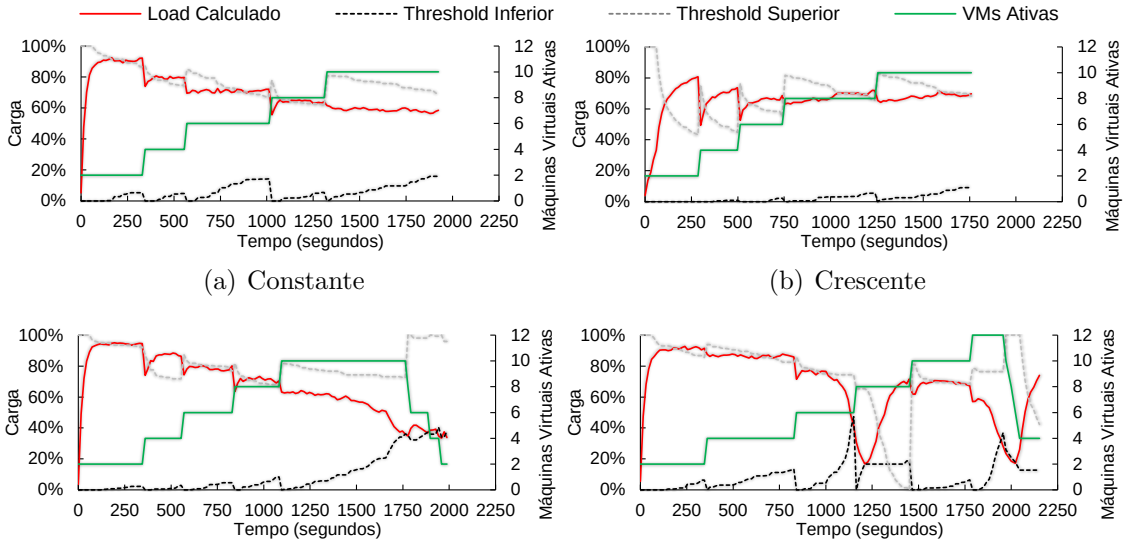
Com o objetivo de gerar apenas uma abordagem definitiva para guiar o funcionamento da técnica Live Thresholding de SelfElastic, foi realizada uma avaliação dos resultados utilizando a técnica chamada Modelo de Soma Ponderada (ou *Weighted Sum Model*) [Triantaphyllou 2000], em que um peso maior significa um melhor desempenho. Os resultados apresentados na Tabela 2 revelaram LT_{ce} como tendo o maior peso somando-se os pesos obtidos para cada comportamento de carga, sendo esta combinação a adotada como abordagem final para a técnica Live Thresholding. Além da perspectiva de Custo, LT_{ce} também obteve o melhor índice do ponto de vista do tempo de execução da aplicação. Foi observado também que a abordagem A_d obteve os piores resultados em termos de desempenho (tempo) pois o *threshold* superior é reiniciado para 100 a cada ação de elasticidade e, dessa maneira, adiando o momento em que a curva de carga do sistema vai atingir novamente o *threshold*.

Tabela 2. Utilizando a métrica de Custo para definir a solução final para a técnica Live Thresholding.

Aplicação	Abordagem								
	LT_{ad}	LT_{ae}	LT_{af}	LT_{bd}	LT_{be}	LT_{bf}	LT_{cd}	LT_{ce}	LT_{cf}
Constante	0,7	0,8	0,9	0,4	0,3	0,6	0,2	1,0	0,5
Crescente	0,3	0,9	0,5	0,2	0,6	0,7	0,4	1,0	0,8
Decrescente	0,8	0,6	0,5	0,3	0,2	0,4	1,0	0,9	0,7
Onda	0,8	0,7	1,0	0,4	0,5	0,3	0,6	0,2	0,9
Total	2,6	3,0	2,9	1,3	1,6	2,0	2,2	3,1	2,9

A Figura 4 apresenta a disponibilidade e utilização de recursos do ambiente

de nuvem durante a execução dos quatro comportamentos de carga. Um ponto importante a ser destacado é que em todas as figuras nota-se que, apesar de comportamentos de carga definidos, os dois *thresholds* sofrem variações. Isso ocorre pois as pequenas variações entre duas observações ocasionam variações nos *thresholds* dependendo se é maior ou menor que 0. Como essas pequenas variações são aplicadas de imediato ao *threshold* correspondente, elas acabam se acumulando; porém, devido a cada um dos comportamentos de carga possuírem uma tendência ao longo do tempo, isso resulta em maiores variações no *threshold* afetado pela tendência da aplicação. A cada nova operação de elasticidade, o *threshold* contrário do qual foi violado retorna para seu valor inicial e as pequenas variações que se acumularam até esse ponto são descartadas.



(a) Constante (b) Crescente

(c) Decrescente (d) Onda

Figura 4. Carga de processamento do sistema e disponibilidade de recursos nas execuções dos quatro comportamentos de carga.

6. Conclusão

Este artigo apresentou SelfElastic, um modelo que explora a ideia de séries temporais da elasticidade proativa e *thresholds* da elasticidade reativa. Tal combinação é responsável por oferecer o conceito chamado de Live Thresholding, o qual representa a principal contribuição de SelfElastic. A ideia é que o usuário não tenha mais que definir os *thresholds* para tirar proveito da elasticidade, os quais são dependentes do tipo de aplicação e da infraestrutura utilizados. Agora os *thresholds* inferior e superior são adaptados a cada atividade periódica de monitoramento, bem como quando uma ação de elasticidade acontece. SelfElastic não só procura atingir um melhor tempo para a execução da aplicação paralela se comparado com a execução não elástica, mas também visa obter um custo igual ou melhor.

Para avaliação do modelo, foram realizados experimentos com uma aplicação paralela iterativa e mestre-escravo num ambiente de nuvem privada. Com o uso de SelfElastic foi possível obter ganhos de desempenho de 50,4% a 59%, na comparação com execuções com recursos fixos. É importante destacar que a técnica Live Thresholding aumenta a reatividade da elasticidade pois na medida que a carga aumenta,

por exemplo, o *threshold* superior diminui indo de encontro com a carga mais rapidamente e, consequentemente, antecipando a adição de novos recursos para atender a demanda da aplicação.

SelfElastic possui algumas limitações que podem ser vistas como oportunidades para trabalhos futuros: (i) apesar de apresentar resultados satisfatórios, cada ação de elasticidade aumenta ou diminui um nó e x VMs (onde x é o número de núcleos do nó); (ii) até o momento, SelfElastic foi avaliado com uma única aplicação. Na exploração de (i), é possível a proposição de técnicas adaptativas para que se tenha um grão maleável de elasticidade. Quanto a (ii), procurou-se explorar diferentes comportamentos da aplicação de modo a testar SelfElastic sob diferentes situações. Mesmo assim, os autores concordam que é necessário avaliar outras aplicações, principalmente aquelas ditas como irregulares.

Agradecimentos

Este trabalho foi parcialmente suportado pelos seguintes órgãos brasileiros de fomento: CAPES, CNPq e FAPERGS.

Referências

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

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

Comanescu, M. (2012). Implementation of time-varying observers used in direct field orientation of motor drives by trapezoidal integration. In *Power Electronics, Machines and Drives (PEMD 2012), 6th IET Int. Conf. on*, pages 1–6. IET.

Copil, G., Moldovan, D., Truong, H.-L., and Dustdar, S. (2013). Sybl: An extensible language for controlling elasticity in cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM Int. Symposium on*, pages 112–119. IEEE.

Dustdar, S., Gambi, A., Krenn, W., and Nickovic, D. (2015). A pattern-based formalization of cloud-based elastic systems. In *Proceedings of the Seventh Int. Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, PESOS '15, pages 31–37, Piscataway, NJ, USA. IEEE Press.

Farokhi, S., Jamshidi, P., Brandic, I., and Elmroth, E. (2015). Self-adaptation challenges for cloud-based applications : A control theoretic perspective. In *10th Int. Workshop on Feedback Computing (Feedback Computing 2015)*. ACM.

Herbst, N. R., Kounev, S., Weber, A., and Groenda, H. (2015). Bungee: An elasticity benchmark for self-adaptive IaaS cloud environments. In *Proceedings of the 10th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 46–56, Piscataway, NJ, USA. IEEE Press.

Imai, S., Chestna, T., and Varela, C. A. (2012). Elastic scalable cloud computing using application-level migration. In *Proceedings of the 2012 IEEE/ACM Fifth Int. Conf. on Utility and Cloud Computing*, UCC '12, pages 91–98, Washington, DC, USA. IEEE Computer Society.

Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. (2010). Performance analysis of high performance computing applications on the Amazon Web Services cloud. In *Proceedings of the 2010 IEEE Second Int. Conf. on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 159–168, Washington, DC, USA. IEEE Computer Society.

Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592.

Mariani, S., Truong, H.-L., Copil, G., Omicini, A., and Dustdar, S. (2014). Coordination-aware elasticity. In *7th IEEE/ACM Int. Conf. on Utility and Cloud Computing (UCC 2014)*, pages 56–63, London, UK. IEEE Computer Society.

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

Nikravesh, A. Y., Ajila, S. A., and Lung, C.-H. (2015). Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *Proceedings of the 10th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 35–45, Piscataway, NJ, USA. IEEE Press.

Rajan, D., Canino, A., Izaguirre, J. A., and Thain, D. (2011). Converting a high performance application to an elastic cloud application. In *Proceedings of the 2011 IEEE Third Int. Conf. on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 383–390, Washington, DC, USA. IEEE Computer Society.

Raveendran, A., Bicer, T., and Agrawal, G. (2011). 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, pages 940–947, Washington, DC, USA. IEEE Computer Society.

Righi, R. d. R., Rodrigues, V. F., Costa, C. A. d., Galante, G., Bona, L. C. E. d., and Ferreto, T. (2016). Autoelastic: Automatic resource elasticity for high performance applications in the cloud. *IEEE Transactions on Cloud Computing*, 4(1):6–19.

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

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

Tanenbaum, A. (2003). *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey, 4th edition.

Triantaphyllou, E. (2000). *Multi-Criteria Decision Making Methodologies: A Comparative Study*, volume 44 of *Applied Optimization*. Springer, Dordrecht.