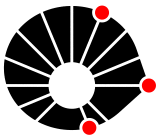
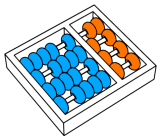




Desenvolvimento de ferramenta para otimização de custo na AWS

Nicholas T. Okita *Tiago A. Coimbra* *Charles B. Rodamilans*
Edson Borin

Relatório Técnico - IC-PFG-18-30
Projeto Final de Graduação
2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Desenvolvimento de ferramenta para otimização de custo na AWS

Nicholas T. Okita* Tiago A. Coimbra† Charles B. Rodamilans‡
Edson Borin§

Resumo

A nuvem computacional viabiliza a execução de programas de alto desempenho sem a necessidade de aquisição de *clusters* e de forma flexível, sendo oferecido diferentes recursos computacionais a preços diferenciados. Neste trabalho exploramos como executar um programa de alto desempenho da área de geofísica utilizando o provedor *Amazon Web Services* (AWS) e o modelo de programação *Scalable Partially Idempotent Task System* (SPITS). Porém, além da execução, também exploramos como minimizar seu custo utilizando algoritmos para escolha das melhores instâncias para nosso programa e as instâncias do mercado *Spot* da AWS. Propomos três novos algoritmos para troca de instâncias e todos foram capazes de ajustar durante tempo de execução as instâncias utilizadas para obtermos melhores custos.

1 Introdução

O modelo de negócios de infraestrutura como serviço (*Infrastructure as a Service* - IaaS) é oferecido para os usuários pelos principais provedores de serviço em nuvem computacional, tais como, *Amazon Web Services* (AWS) da Amazon [8], *Azure* da Microsoft [9] e Google Cloud Platform [14], da Google. Neste modelo é permitido aos usuários a instanciação de máquinas virtuais com diferentes combinações de *hardware*, como número de núcleos de processamento e quantidade de memória RAM, e a configuração com sua própria pilha de *software*.

Neste relatório documentamos o estudo sobre o serviço de computação em nuvem AWS e suas limitações, como usamos a plataforma SPITS na nuvem computacional e por fim implementações de algoritmos para otimização do custo de execução e seus resultados.

*Aluno, Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
†Co-Orientador, Centro de Estudos de Petróleo, Universidade Estadual de Campinas, 13083-896 Campinas, SP.
‡Colaborador, Faculdade de Computação e Informática (FCI), Universidade Presbiteriana Mackenzie, 01302-000 São Paulo, SP.
§Orientador, Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

1.1 Motivação e Objetivos

A computação de alto desempenho em nuvem é uma forte alternativa para a aquisição de *clusters*. Além de oferecer maior flexibilidade na execução, dado que no modelo IaaS o usuário escolhe as configurações de *hardware* e o *software* instalado nas máquinas, em algumas cargas de trabalho a nuvem pode oferecer um custo inferior ao da execução em *clusters* [15].

Entretanto, para usufruirmos da possibilidade de reduções de custo ao utilizar a nuvem, enfrentamos dificuldades na seleção de *hardware*, haja vista que diferentes configurações possuem diferentes custos. Não somente isso, mas existem também várias formas de se contratar o serviço (afetando seu custo e funcionalidade) e opções para armazenamento de dados, além de outros desafios que serão explicadas na subseção 1.2.

As dificuldades encontradas para redução de custo motivaram este trabalho cujos objetivos são: (i) a execução de um programa de alto desempenho distribuído na nuvem AWS; (ii) a minimização do custo de execução deste programa.

1.2 Amazon Web Services

A *Amazon Web Services* (AWS) é uma plataforma de serviços de computação em nuvem oferecida pela *Amazon* que provê serviços de computação, armazenamento, banco de dados, *machine learning*, etc. O serviço que usaremos é voltado para computação e é chamado *Elastic Computer Cloud* (EC2), o qual disponibiliza máquinas virtuais com diferentes configurações sob diferentes custos dependendo da configuração escolhida e do tipo de contrato.

1.2.1 Elastic Computer Cloud

A precificação dos serviços varia de acordo com o serviço contratado, o contrato firmado, a região e zona de disponibilidade escolhidas. No caso do serviço que usaremos (EC2), são disponibilizados três contratos para adquirir uma máquina virtual. Em instâncias *On-Demand* alugamos uma configuração de máquina virtual e pagamos por hora de uso; em instâncias *Reserved* é realizada uma negociação com a *Amazon* e contrata-se uma máquina virtual por um período extenso de tempo (meses, por exemplo), sendo que é oferecido um custo por hora reduzido devido ao compromisso de longo termo; por fim em instâncias *Spot* o usuário contrata recursos computacionais que não estão sendo utilizados pelo provedor, podendo pagar taxas mais baixas - reguladas por oferta e demanda - porém correndo o risco de terminação pelo provedor quando esses recursos forem necessários.

Na Tabela 1 são mostradas algumas configurações disponíveis na zona de disponibilidade *us-east-1a* e seus respectivos custos ao usar o contrato de instâncias *On-Demand*. Nela temos o nome da instância seguido do número de núcleos virtuais disponíveis para esta instância (*threads*), quantidade de memória RAM, o custo por hora na zona de disponibilidade *us-east-1a* para o contrato *On-Demand* e por fim a finalidade da instância de acordo com a provedora de serviço. Por exemplo a instância m5.12xlarge possui 48 *threads* do processador Xeon Platinum 8175 2.5GHz com 192GB de RAM voltada para uso geral e custa US$2,304 por hora de uso.

Tabela 1: Instâncias AWS

Instância	vCPUS (número de núcleos virtuais)	RAM (GB)	Custo (USD/h)	Otimização
c4.4xlarge	16(Xeon E5-2666 v3 Haswell)	30	0,796	Computação
c4.8xlarge	32(Xeon E5-2666 v3 Haswell)	60	1,591	Computação
c5.4xlarge	16(Xeon Platinum 8124 3GHz)	32	0,680	Computação
c5.9xlarge	36(Xeon Platinum 8124 3GHz)	72	1,530	Computação
c5.18xlarge	72(Xeon Platinum 8124 3GHz)	144	3,060	Computação
d2.4xlarge	16(Xeon E5-2676 v3 Haswell)	122	2,760	Armazenamento
d2.8xlarge	36(Xeon E5-2676 v3 Haswell)	244	5,520	Armazenamento
m4.4xlarge	16(Xeon E5-2676 v3 Haswell)	64	0,800	Uso geral
m4.10xlarge	40(Xeon E5-2676 v3 Haswell)	160	2,000	Uso geral
m5.4xlarge	16(Xeon Platinum 8175 2.5GHz)	64	0,768	Uso geral
m5.12xlarge	48(Xeon Platinum 8175 2.5GHz)	192	2,304	Uso geral
m5.24xlarge	96(Xeon Platinum 8175 2.5GHz)	384	4,608	Uso geral
r4.4xlarge	16(Xeon E5-2686 v4 Broadwell)	122	1,064	Memória
r4.8xlarge	32(Xeon E5-2686 v4 Broadwell)	244	2,128	Memória
r4.16xlarge	64(Xeon E5-2686 v4 Broadwell)	488	4,256	Memória

1.2.2 Zonas de Disponibilidade

A *Amazon* possui *datacenters* em diferentes regiões no mundo, sendo que cada região é isolada geograficamente de outra, isto é, instâncias de uma região não podem se comunicar diretamente (por IP privado) com instâncias de outras regiões. Porém, dentro das regiões temos conexões de baixa latência conectando os *datacenters*, separando eles nas chamadas zonas de disponibilidade. Podemos criar instâncias em quaisquer zonas de disponibilidade de uma região e elas poderão se comunicar entre si livremente.

As zonas de disponibilidade são identificadas por uma letra após o nome da região, por exemplo: **us-east-1a** refere-se à zona de disponibilidade 'a' na região *us-east-1* (que representa Virgínia do Norte).

1.2.3 Armazenamento

Ademais, ainda existem gastos relacionados ao tipo de dispositivo de armazenamento alocado para a instância (chamado de *Elastic Block Storage* (EBS)), sendo disponibilizados quatro tipos com diferentes custos para diferentes desempenhos (medido em *Input Output Operations per Second* (IOPS) para os SSDs ou *throughput* em MB/s para os HDDs) [2], assim como mostrado na Tabela 2.

O dispositivo utilizado como padrão para instanciação de novas máquinas atualmente é o gp2, apresentando desempenho superior às opções de HDD. Para os dispositivos gp2, st1 e sc1 o desempenho varia com o espaço alocado, sendo que quanto maior o tamanho do dispositivo, maior será seu desempenho. Por exemplo, um armazenamento do tipo gp2 com 100GB alocados possui 300IOPS de desempenho (max(100 IOPS, 3IOPS/GB $\times$ 100GB)), enquanto um armazenamento gp2 com 20GB alocados possui 100 IOPS. Por outro lado, o dispositivo io1 tem sue desempenho definido pelo usuário. Descrevemos na Equação 1 a relação de custos e tempos para decidir quando o uso do dispositivo gp2 oferece melhor

Tabela 2: Dispositivos de Armazenamento EBS

Tipo de dispositivos de armazenamento	Desempenho (IOPS/throughput)	Custo
General Purpose SSD (gp2)	max(100 IOPS, 3 IOPS/GB)	0,1/(G×mês)
Provisioned IOPS SSD (io1)	IOPS definido pelo usuário	0,125/(GB×mês) + 0,065/(IOPS×mês)
Throughput Optimized HDD (st1)	40MB/s/TiB	0,045/(GB×mês)
Cold HDD (sc1)	12MB/s/TiB	0,025/(GB×mês)

valor.

$$\left(\frac{t_{gp2}}{t_{io1}} - 1\right) c_{inst} + \left(\frac{t_{gp2}}{t_{io1}} - \alpha\right) c_{gp2} - \beta \leq 0 \quad (1)$$

Na Equação 1 temos que t_{gp2} e t_{io1} são os tempos que a instância permanecerá ligada com o dispositivo do tipo gp2 e io1, respectivamente. c_{inst} é o custo por hora da instância selecionada assim como c_{gp2} é o custo por hora do dispositivo gp2 de tamanho definido pelo usuário. A constante α representa a razão de custo por hora do dispositivo io1 pelo dispositivo gp2 considerando somente o armazenamento e que ambos possuem o mesmo tamanho, atualmente (como mostrado na Tabela 2) este valor é $\alpha = 1.25$, enquanto a constante β representa o custo por hora da escolha de IOPS para o dispositivo io1.

Como exemplo de uso desta equação, dada uma aplicação CPU *bound* com um longo tempo de execução, tal que $t_{io1} \approx t_{gp2}$, vemos que $\frac{t_{gp2}}{t_{io1}} \approx 1$. Além disso, sabemos pela Tabela 2 que $\alpha = 1.25$ e consideramos que alocamos um dispositivo de 200GB tanto para gp2 quanto para io1 ($c_{gp2} \approx 0.03\frac{USD}{h}$), e no caso do dispositivo io1 escolhemos 1000 IOPS (portanto temos $\beta \approx 0.1\frac{USD}{h}$). Por fim, aplicando estes valores na Equação 1 temos: $(0 - 0.25 \times 0.03 - 0.1)\frac{USD}{h} = -1.075\frac{USD}{h} \leq 0$. Concluindo que para este exemplo o uso do dispositivo gp2 oferece melhor custo.

Por fim, as opções com HDD (st1 e sc1) são mais voltadas para instâncias que armazenarão grandes volumes de dados por longos períodos de tempo, sem a necessidade de alto desempenho de IO.

Outrossim, utilizamos os serviços de armazenamento da AWS para armazenar programas, dados e resultados de execução. São disponibilizados dois serviços além das opções do EBS: *Simple Storage Service* (S3) e *Elastic File System* (EFS).

O serviço S3 é voltado para o armazenamento de objetos em *buckets* [6]. Podemos armazenar quaisquer objetos com menos de 5TB e quantos objetos quisermos dentro de um *bucket*. Além disso, seu acesso é controlado pelo usuário criador ou administrador da conta.

Para utilizar um objeto em um *bucket* é necessário fazer seu *download* para a instância, não é possível acessar diretamente o S3 como um *file system*. Seu custo é composto pelo número de *bytes* transferidos de um *bucket* para uma instância ou um computador local, pela quantidade de dados armazenados em *buckets* [7].

Por outro lado o serviço EFS oferece um sistema de arquivos que pode ser compartilhado entre diferentes instâncias [3]. Por se tratar de um sistema de arquivos, podemos acessá-lo tanto para escrita quanto para leitura assim como fazemos com dispositivos montados nas instâncias, tornando seu uso para sistemas distribuídos e *clusters* mais simples do que o

S3. Entretanto, o EFS apresenta um custo superior ao outro serviço de armazenamento da AWS, sendo cerca de dez vezes mais caro por *gigabyte* armazenado [4].

1.2.4 Spot

Instâncias Spot são instâncias de máquinas virtuais que são oferecidas por custo menor mas com menos garantias. Estas instâncias podem sofrer flutuação de custo, encerramento precoce por parte do provedor e indisponibilidade. Ainda, neste tipo de contrato a diferença de custos entre zonas de disponibilidade é significativo, como visto na Figura 1, na qual comparamos o custo da instância c5.18xlarge em diferentes zonas de disponibilidade no decorrer do mês de abril na região *us-east-1*. Nesta figura, vemos que a variação de custo entre zonas de disponibilidade pode chegar a 30%, como ocorreu no dia 6 de abril de 2018 entre a zona *us-east-1c* e *us-east-1f*.

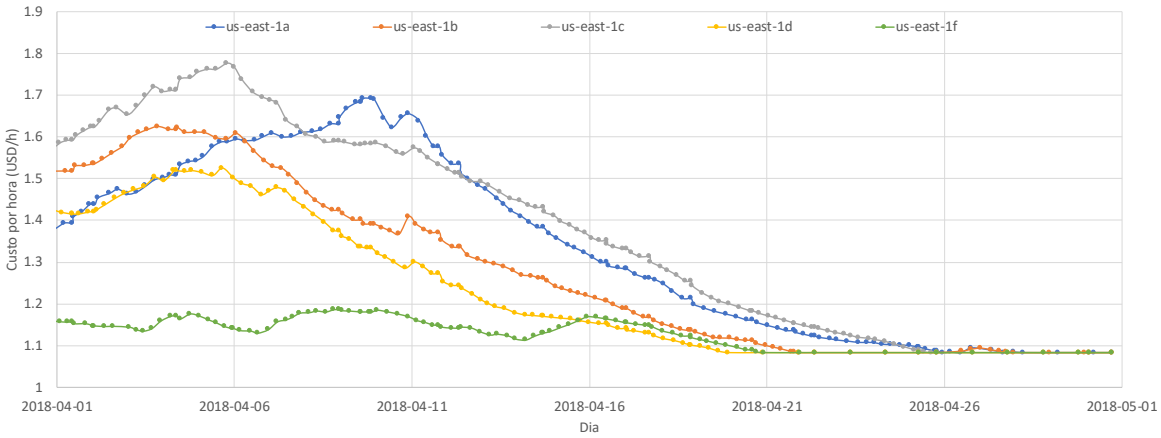


Figura 1: Variação do custo da instância *c5.18xlarge* no modelo *Spot* em diferentes zonas de disponibilidade ao longo do mês de abril de 2018

No contrato de instâncias *Spot*, o usuário define um valor máximo a ser pago pela instância. O momento que o custo da máquina superar este valor máximo ou a *Amazon* necessitar de mais recursos computacionais, a instância é terminada e o usuário deve então fazer um novo pedido. Vale ressaltar que o custo das máquinas varia no decorrer do tempo (como visto na Figura 1) de acordo com oferta e demanda.

Portanto, devido a essas condições no contrato *Spot*, concluímos que nosso programa deve ser tolerante a falhas e possuir provisionamento dinâmico de recursos. A tolerância a falhas é necessária para o programa continuar executando na situação de encerramento de instâncias pela AWS. O provisionamento dinâmico permite a adição de novas máquinas virtuais durante a execução do programa, o que viabiliza o aumento do poder computacional ou migração da computação para novas máquinas virtuais.

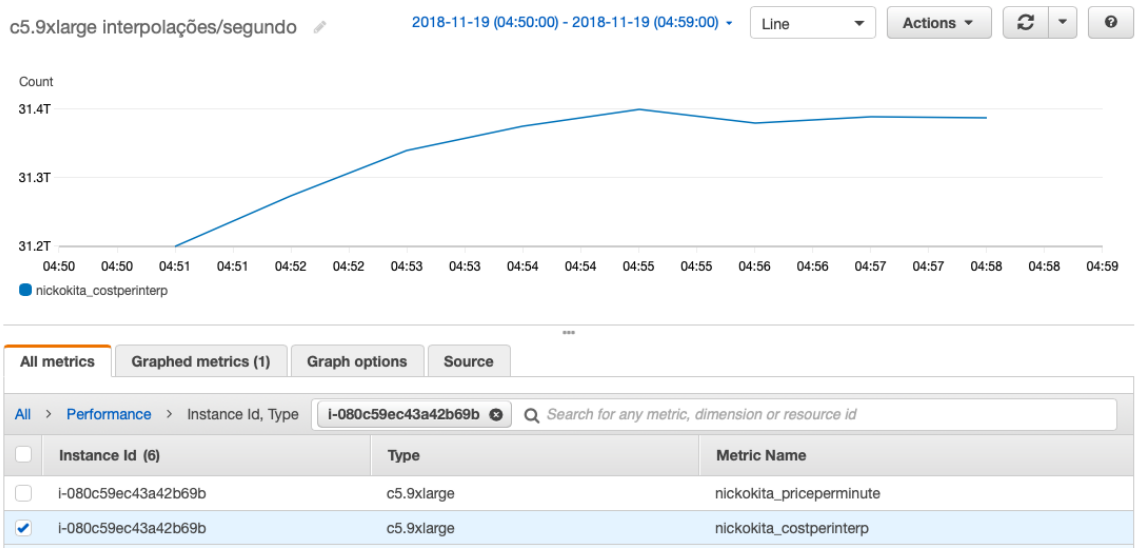


Figura 2: Visualização gráfica de métrica no *CloudWatch*

1.2.5 Cloudwatch e Lambda

CloudWatch é uma plataforma da AWS para monitoramento e gerenciamento de instâncias e serviços da AWS [1]. Resumidamente, com esta plataforma podemos coletar métricas e relatórios de uso de nossas instâncias, analisá-las com gráficos e aplicar medidas corretivas (como auto escalabilidade baseada em uso de CPU). Além disso, existe a possibilidade de ativar alarmes alertar quando alguma métrica está se comportando de forma indesejável. Apesar de existirem mais funcionalidades, essas são as que interessam neste trabalho.

O *CloudWatch* já oferece algumas métricas, por exemplo a medição do uso de CPU e de memória RAM de instâncias EC2. Entretanto, a plataforma permite a implementação de nossas próprias métricas. Dessa forma, podemos extrair informações relevantes para nossa aplicação utilizando a própria plataforma da AWS, somente com a necessidade de implementação do envio dessas métricas.

Como exemplo para visualização das métricas colocamos na Figura 2 a visualização em forma de gráfico de linha de uma métrica que implementamos. Neste exemplo escolhemos a métrica de nome nickokita_costperinterp (métrica de interpolações por US$, que será explicada na seção 2.3.1) para a instância de id i-080c59ec43a42b69b durante o período de 4:50 a 4:59 do dia 19 de novembro de 2018.

Além de permitir visualização de nossas métricas, a utilização da plataforma *CloudWatch* permite o uso do serviço *Lambda* para aplicarmos nossas ações corretivas.

Lambda é um serviço da AWS que executa códigos (*scripts Python* ou *NodeJS* por exemplo) sem a necessidade de alocação de recurso computacional por parte do usuário [5]. O serviço pode ser utilizado para execução de código em resposta a eventos, como os gerados pelo *CloudWatch*, por exemplo.

A vantagem de utilização desses serviços é a automatização de monitoramento e respostas. Ou seja, além de monitorar recursos utilizando o *CloudWatch*, podemos gerar eventos

para serem processados pelo *Lambda* automaticamente.

1.3 SPITS

O modelo de programação *Scalable Partially Idempotent Task Systems* (SPITS) foi desenvolvido no laboratório *High Performance Geophysics* (HPG) e é utilizado para implementar programas em sistemas distribuídos utilizando tarefas idempotentes [12] [11] [10]. Sendo que, uma tarefa idempotente é uma tarefa que não depende do resultado de outras e ao ser executada múltiplas vezes retorna sempre o mesmo resultado.

O modelo SPITS envolve a divisão do programa em três partes independentes entre si: *Job Manager* (JM), *Worker* (WK) e *Committer* (CO). O *Job Manager* é responsável pela geração de tarefas, as quais são repassadas para os *Workers* que processam essas tarefas e enviam o resultado para o *Committer* que serializa e armazena o resultado da computação, assim como exemplificado na Figura 3.

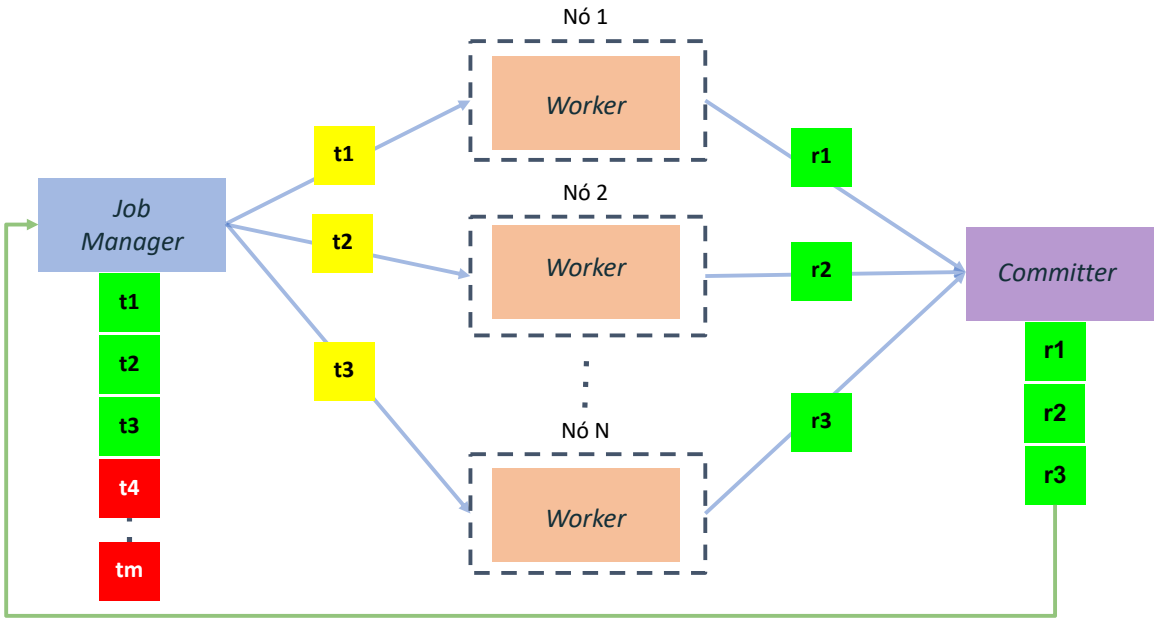


Figura 3: Fluxo de geração e execução de tarefas no SPITS

O paralelismo no SPITS se dá nos *Workers*, criamos um *Worker* por máquina e enviamos tarefas geradas pelo *Job Manager* para cada um deles. Esse modelo permite por consequência o provisionamento dinâmico de recursos, haja vista que basta a inserção de um novo *Worker* em tempo de execução para aumentarmos a quantidade de recursos disponíveis para a execução. Permite também arquiteturas heterogêneas, pois podemos ter *Workers* com CPUs ou GPUs e cada Worker implementa seu código (desde que a entrada e a saída possuam o mesmo formato). Por fim possui tolerância a falhas porque se um *Worker* for desativado por alguma falha, a propriedade de idempotência das tarefas permite que elas sejam redistribuídas entre os *Workers* que ainda funcionam e o processo continue corretamente.

Escolhemos o modelo de programação SPITS pois ao trabalhar com instâncias *Spot* lidamos com o risco de termos nossas instâncias encerradas abruptamente pelo provedor de serviços. Além disso, como queremos otimizar o custo escolhendo as melhores instâncias, iniciaremos e encerraremos instâncias no decorrer da execução, portanto provisionamento dinâmico é um recurso necessário e está disponível no SPITS.

Além destes requisitos necessários, o SPITS oferece alguns recursos que são desejáveis para nosso programa. Não somente é uma plataforma simples de programar, utilizamos o *runtime* PY-PITS [11] o qual necessita somente de *Python* para ser executado, um pacote disponível em todas distribuições *Linux* atuais.

1.3.1 SPITS na nuvem

Para utilizar o SPITS na nuvem não foi necessário nenhuma alteração no código do programa desenvolvido utilizando o modelo de programação nem no *runtime* PY-PITS, haja vista que o modelo já é voltado para sistemas distribuídos. Quanto a infraestrutura na nuvem, foi necessário somente configurar as instâncias para permitir comunicação entre si por quaisquer portas na rede local (caracterizada pela região). Como a execução do PY-PITS depende de um diretório compartilhado, utilizamos o serviço EFS para compartilhar o sistema de arquivos entre as diferentes máquinas no sistema.

Utilizamos o modelo mostrado na Figura 4 para executar programas SPITS na nuvem computacional. Neste modelo utilizamos uma instância do tipo *On-Demand* para ambos *Job Manager* e *Committer*, várias instâncias *Spot* para os *Workers* e o sistema de arquivos EFS para armazenar os dados (tanto para leitura dos dados, quanto para escrita dos resultados).

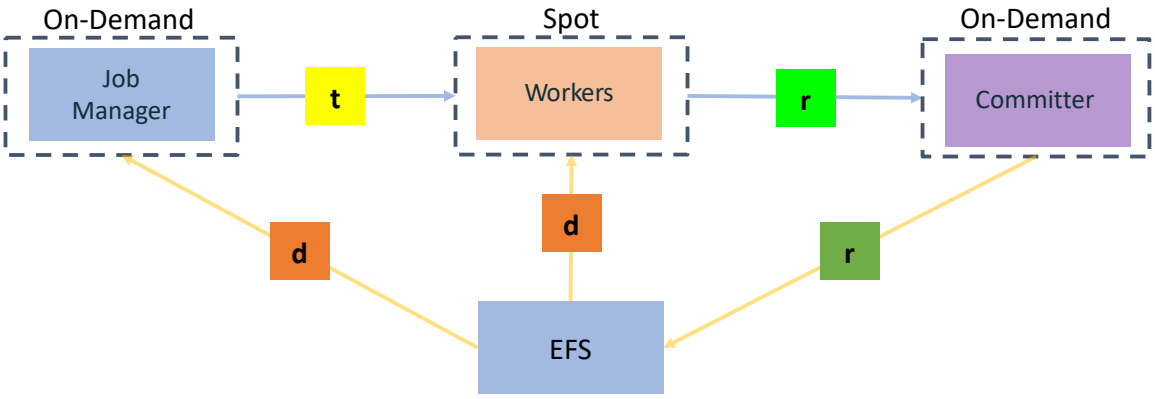


Figura 4: Fluxo de informações entre o EFS e os módulos do SPITS

Na Figura 4, vemos o fluxo de informações entre os módulos do SPITS. O *Job Manager* e os *Workers* leem o dado de entrada a partir do EFS, em seguida o *Job Manager* envia tarefas aos *Workers*, que por sua vez processam e enviam o resultado para o *Committer* que salva o resultado consolidado de volta no EFS.

1.3.2 Reconfiguração PY-PITS na Spot

Uma das pastas dentro do diretório compartilhado do PY-PITS possui arquivos com o endereço e a porta dos *Workers* na rede, o qual é utilizado pelo *Job Manager* e pelo *Committer* para se conectar aos *Workers* a fim de enviar e receber tarefas e resultados. Se uma instância falha, seu arquivo contendo seu endereço e porta permanece na pasta e eventualmente o *Job Manager* tenta conexão, porém como a instância não responde, temos que aguardar o *timeout* do TCP para tentar conexão com outras instâncias (2 minutos).

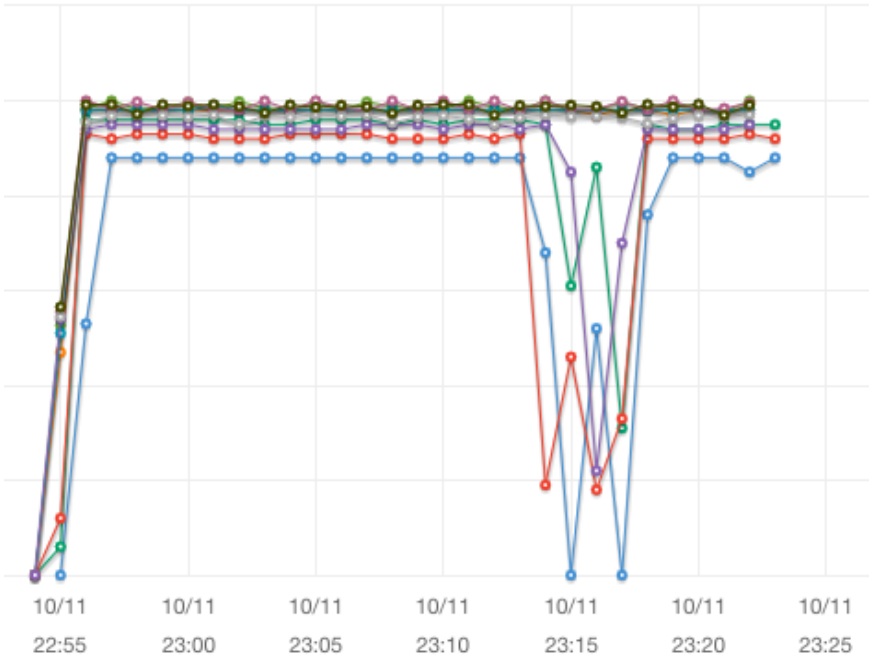


Figura 5: Uso de CPU no decorrer do tempo em instâncias

Este problema de *timeout* acarretou em problemas de desempenho como visto na Figura 5, em que uma instância é encerrada pela AWS e causa perda de desempenho em outras instâncias. Além disso, percebemos que algumas execuções nunca encerravam e encontramos que o problema era o *timeout* na espera de envios e recebimentos de pacotes.

Para corrigir o problema, simplesmente colocamos *flags* para o *runtime* PY-PITS reduzir todos *timeouts* para 5 segundos. Isto é, ao tentar conectar em um nó desativado o *Job Manager* aguarda somente 5 segundos até tentar conectar no próximo nó. Este tempo foi suficiente para resolver o problema de desempenho e não causou instabilidades na execução.

2 Metodologia

Nesta seção serão discutidas as propostas para otimização de custo na AWS bem como os experimentos realizados.

2.1 Ferramental

Para a realização dos experimentos que serão discutidos nas próximas subseções utilizamos o sistema mostrado na Figura 4 com uma instância *c5.4xlarge* como *Job Manager* e *Committer* e as as instância mostradas na Tabela 1 como *Workers*.

Foi utilizado um programa de processamento sísmico de alto desempenho que tem como finalidade a computação dos parâmetros do método *Non-hyperbolic Common Reflective Surface*, ou *NCRS* [13]. Este programa foi implementado utilizando o modelo SPITS e utilizamos o *runtime* PY-PITS [11] para sua execução. Além disso, a implementação do programa utiliza a heurística *Differential Evolution* (*DE*) [18] para realizar a busca dos parâmetros.

O programa *NCRS* foi compilado utilizando o compilador *gcc* versão *5.4.0* com a *flag* de compilação *-O3*, enquanto que o motor *PY-PITS* foi executado utilizando *python* versão *3.5.2*. A entrada do programa foi um dado sísmico de aproximadamente 1,3 GB e escolheu-se para o *DE* uma população de tamanho 31 iterando por 31 gerações.

Em cada instância utilizamos um disco de 20GB do tipo io1 com 1000 IOPS, haja vista que calibramos nossos testes para serem de curta duração, sendo mais vantajoso a utilização de um disco mais rápido. Este disco foi inicializado a partir de uma imagem baseada em Ubuntu 16.04, o qual possui diversos pacotes já instalados (*awscli*, *binutils*, *cloud-utils*, *efs-utils*, *gcc*, *make*, *python3-pip*, *sshpass*, *unzip* e *zip*), uma pasta para o EFS montado e uma cópia do dado sísmico de entrada. O programa e os resultados foram armazenados no EFS.

Os *scripts* que desenvolvemos (mostrados na Seção 2.3.1) foram implementados em *python 3.6* e são executados utilizando o serviço *Lambda*. Além disso utilizamos o serviço *CloudWatch* para monitorar as instâncias e armazenar informações durante a execução para serem utilizadas pelo *Lambda*.

Por fim, por simplicidade, consideramos somente o custo das instâncias ao computar o custo total de execução, desconsiderando, portanto, o custo dos discos EBS, do EFS, das transferências de *bytes* entre zonas de disponibilidade e dos serviços *Lambda* e *CloudWatch*. Vale destacar que estes custos são pequenos quando comparados aos custos das máquinas virtuais.

2.2 Curva Pareto

Uma das maiores dificuldades para otimização de custo é a escolha adequada de máquinas para determinada aplicação. Para resolver este problema a curva Pareto é utilizada como guia para escolher máquinas que apresentem melhor custo ou tempo de execução [16].

A curva Pareto é composta por pontos p_i, tal que um ponto p_i não seja dominado por nenhum ponto p_j. No nosso caso, consideramos o conjunto de pontos P como pares de coordenadas (x,y) no plano cartesiano R^2. E definimos o conjunto de pontos P' que domina um ponto p_i como:

$$P'(p_i) = \{p \in P : x_p > x_{p_i} \wedge y_p > y_{p_i}, p \neq p_i\} \tag{2}$$

Logo, consideramos que um ponto está na curva Pareto P_p se:

$$P_p = \{p \in P : P'(p) = \emptyset\} \tag{3}$$

Para nosso trabalho usamos um plano R^2 tal que o eixo das abscissas (x) representa o tempo de execução do programa e o eixo das ordenadas (y) representa o custo de execução do programa. Nosso conjunto de pontos P_I representa os diferentes tipos de máquina, mostrando então para aquela uma dada instância qual seu tempo de execução e qual seu custo. Por fim, definimos que uma instância I_i domina outra I_j quando I_i apresenta ambos tempo de execução e custo menores que o de I_j, portanto redefinimos a Equação 2 como:

$$P'(I_i) = \{I \in P_I : x_I > x_{I_i} \wedge y_I > y_{I_i}, I \neq I_i\} \tag{4}$$

Sendo assim, a curva Pareto é composta por um conjunto de pontos P_{p_I} tal que nenhum ponto seja dominado por quaisquer outros pontos, isto é:

$$P_{p_I} = \{p \in P_I : P'(p) = \emptyset\} \tag{5}$$

Com conhecimento dessas relações podemos escolher as instâncias que executam o programa de forma mais eficiente, isto é, podemos traçar a curva Pareto e encontrar instâncias que executam em menor período de tempo ou com menor custo.

Vale ressaltar que como estamos trabalhando com instâncias *Spot* esse gráfico é variável com o tempo, já que o custo de um tipo de máquina pode variar. Entretanto, para nossos experimentos desconsideramos essa possibilidade, visto que o curto período de tempo que estamos executando não se mostrou suficiente para variar os custos de forma a alterar a curva.

2.3 Otimização de custo

A curva Pareto em nossos resultados experimentais mostrou que de fato existe uma grande diferença no custo de execução entre diferentes tipos de máquinas. Logo, a escolha de máquinas é o problema mais significativo para a minimização de custo da execução do programa. Ademais, vimos que as instâncias *Spot* oferecem desempenho muito semelhante às instâncias *On-Demand* porém com um custo cerca de três vezes menor, portanto de fato utilizar a arquitetura de SPITS na nuvem com *Workers* do tipo *Spot* pode representar uma redução significativa de custo.

Dessa forma, escolhemos otimizar o custo de execução utilizando instâncias *Spot* como *Workers*, sendo a proposta de solução selecionar as melhores instâncias para o programa dinamicamente. A proposta dinâmica é interessante pois podemos ter alterações de custo no decorrer do tempo (tendo em vista o mercado de instâncias *Spot*) e permite que executemos a aplicação já encaminhando para o resultado desejado sem a necessidade de conhecer características nem das máquinas e nem da aplicação previamente.

2.3.1 Políticas de troca

A proposta de solução foi inicializar o processo com um grupo pré definido de máquinas (por exemplo, as instâncias mostradas na Tabela 1). Extraímos informações sobre o desempenho da aplicação durante sua execução (medido em interpolações por segundo para o programa NCRS) e o custo por hora de cada máquina, medindo então sua relação de

desempenho por custo (no caso do programa NCRS em interpolações por US$ [15]). Por fim, com a relação de desempenho por custo podemos selecionar as instâncias que propiciam o menor custo para a execução da aplicação, encerrando as outras.

Uma primeira versão deste algoritmo de ajuste dinâmico é mostrado no Algoritmo 1 e foi implementada por Okita no trabalho Otimização automática do custo de processamento de programas SPITS na AWS [17]. Esta versão considera apenas trocas em intervalos regulares de tempo e uma instância de cada vez, trocando a instância que apresenta pior medida de interpolações por US$ pela instância que apresenta a melhor medida. Consideramos que houve convergência quando a instância que apresenta este pior resultado é a mesma da instância que apresenta o melhor resultado.

Além disso, em cada troca escolhemos a zona de disponibilidade que possui menor custo para a instância desejada. Entretanto neste exemplo não consideramos nem casos de instâncias com medidas de desempenho por custo muito próximo a ponto de não valer a pena a troca, nem a possibilidade de trocar a zona de disponibilidade de uma instância se o custo dela aumentar na zona atual e ficar inferior em outra.

Algorithm 1 Algoritmo de Troca

```
1: L: Lista de tuplas (id, tipo, custo benefício)   ▷ Em ordem cresc. de interpolações por US$
2: procedure TROCA DE INSTÂNCIA(L)
3:     if L[1].tipo ≠ L[len(L)].tipo then
4:         z ← zona de disponibilidade de menor custo para instância L[len(L)].tipo
5:         crie uma instância nova do tipo L[len(L)].tipo em z
6:         encerre a instância L[1].id
```

Para nossos parâmetros de entrada o programa executava em no máximo cerca de uma hora, isto é, o tempo não era suficiente para existir problema com alterações de preços nas zonas de disponibilidade. Porém o problema das medidas semelhantes ainda existia, por conseguinte criamos uma versão revisada deste algoritmo - a qual está mostrada no Algoritmo 2 - que considera o desvio padrão para a escolha das melhores instâncias. Isto é, não consideramos somente o desempenho imediato, mas sim o histórico do desempenho da instância no decorrer da execução da aplicação, computamos então a média e o desvio padrão deste histórico de desempenho. Consideramos que uma instância X apresenta melhor custo por desempenho que uma instância Y se:

$$\frac{(\overline{M}_X - \sigma_{M_X})}{c_X} > \frac{(\overline{M}_Y + \sigma_{M_Y})}{c_Y} \tag{6}$$

Sendo que $\overline{M}_X$ representa a média das medidas de desempenho da instância X, σ_{M_X} representa o desvio padrão dessas medidas e c_X representa o custo por hora da instância X. O significado das variáveis é análogo para Y. Os valores de $\overline{M}_X$ e σ_{M_X} são calculados na instância e enviados para o *CloudWatch*, permitindo que o *script Lambda* apenas tome decisões de troca de instâncias.

Por simplicidade de notação, nos próximos parágrafos assumiremos que:

$$\mu_X = \frac{\overline{M}_X}{c_X} \tag{7}$$

$$\psi_X = \frac{\sigma_{M_X}}{c_X} \tag{8}$$

Sendo que μ_X é a média das medidas de custo benefício para a instância X e ψ_X seu desvio padrão. No Algoritmo 1 podemos substituir a expressão custo benefício por μ_X sem perda de significado. Além disso, podemos reescrever a Equação 6 usando essa notação:

$$\mu_X - \psi_X > \mu_Y + \psi_Y \tag{9}$$

Algorithm 2 Algoritmo de Troca Revisado

```
1: L: Lista de tuplas (id, tipo, μ_id, ψ_id)          ▷ Em ordem cresc. de μ_id
2: procedure TROCA DE INSTÂNCIA(L)
3:     if L[1].tipo ≠ L[len(L)].tipo then
4:         if L[1].μ + L[1].ψ < L[len(L)].μ − L[len(L)].ψ then
5:             z ← zona de disponibilidade de menor custo para instância L[len(L)].tipo
6:             crie uma instância nova do tipo L[len(L)].tipo em z
7:             encerre a instância L[1].id
```

Neste algoritmo ainda trocamos apenas uma instância a cada período de tempo, consequentemente para atingir convergência na escolha de instâncias levamos pelo menos o intervalo de trocas multiplicado pelo número de máquinas. Surge então a questão de troca de mais de uma instância a cada período que o algoritmo é executado, implementamos então uma nova versão do algoritmo de trocas (mostrado no Algoritmo 3). Nesta versão escolhemos uma porcentagem P de instâncias para serem trocadas a cada período de tempo, consequentemente, se não fizermos escolhas erradas, a convergência ocorrerá após $100/P$ iterações.

Algorithm 3 Algoritmo de Troca em Lotes

```
1: L: Lista de tuplas (id, tipo, μ_id, ψ_id)          ▷ Em ordem cresc. de μ_id
2: P: Porcentagem de instâncias para serem trocadas    ▷ Em valores de 0 a 1
3: procedure TROCA DE INSTÂNCIA(L, P)
4:     i = 1 ; lim = max(len(L) * P, 1)
5:     for i ≤ len(L) * P do
6:         if L[i].tipo ≠ L[len(L)].tipo then
7:             if L[i].μ + L[i].ψ < L[len(L)].μ − L[len(L)].ψ then
8:                 z ← zona de disponibilidade de menor custo para instância L[len(L)].tipo
9:                 crie uma instância nova do tipo L[len(L)].tipo em z
10:                encerre a instância L[i].id
```

Vale ressaltar que trocar apenas uma instância é uma opção mais segura com relação a escolhas erradas, haja vista que durante momentos iniciais da execução a instância que apresenta melhor custo benefício pode não ter ainda reportado seu desempenho. Porém, é importante balancear o tempo de convergência (consequentemente a redução de custo) com as perdas causadas por decisões erradas.

Quanto à otimização de desempenho (como tempo de execução) ao invés de custo, basta substituirmos nos algoritmos mostrados anteriormente μ_{id} e ψ_{id} por $\overline{M}_{id}$ e σ_{M_X}, respectivamente.

2.4 Otimização de custo/benefício

Apesar de considerarmos a nuvem contendo recursos computacionais ilimitados, isso não acontece na prática. O serviço da AWS possui limitação em quantas máquinas virtuais podemos ter instanciadas em um momento. Tendo isso em vista, todos algoritmos aqui mostrados consideram essa restrição no número de instâncias, logo nunca aumentamos esta quantidade além das máquinas inicialmente criadas.

Tendo essa restrição em vista, implementamos um algoritmo que balanceia custo e tempo de execução. Isto é, desejamos encontrar a instância que além de apresentar bom custo para a tarefa também realize a tarefa em um menor período de tempo, haja vista que, algumas decisões de redução de custo não são suficientes para compensar a perda de desempenho que pode ocorrer consequentemente, especialmente considerando que temos um número fixo de máquinas.

Essa proposta de balanceamento de custo e tempo de execução foi implementada como o Algoritmo 4. Neste algoritmo normalizamos as medidas de desempenho (em interpolações por segundo) e as medidas de custo (em interpolações por US$) e somamos ambas dado um peso de entrada (w_{sec} para a medida de desempenho, w_{usd} para a medida de custo). Definimos as medidas com pesos usando η e *phi* da forma mostrada pelas equações 10 e 11:

$$\eta_X = w_{sec} \times \frac{\overline{M}_X}{\overline{M}_{max}} + w_{usd} \times \frac{\mu_X}{\mu_{max}} \tag{10}$$

$$\phi_X = w_{sec} \times \frac{\sigma_{M_X}}{\overline{M}_{max}} + w_{usd} \times \frac{\psi_X}{\mu_{max}} \tag{11}$$

Sendo que μ_{max} representa a melhor medida de desempenho por custo encontrado na lista de instâncias, assim como $\overline{M}_{max}$ representa a melhor medida de desempenho na lista de instâncias. O significado das outras variáveis é o mesmo das equações anteriores. Vale ressaltar que não existem regras para a definição dos pesos w_{usd} e w_{sec}. Espera-se que estes pesos sejam definidos pelo usuário, com base no quanto ele está disposto a pagar por ganho de desempenho.

Analogamente à equação 9, uma instância X será considerada melhor que uma instância Y se:

$$\eta_X - \phi_X > \eta_Y + \phi_Y \tag{12}$$

Vale destacar que se usarmos $w_{sec} = 0$ e $w_{usd} = 1$ nas equações 10 e 11 temos a mesma definição das equações 7 e 8. Ou seja, o algoritmo de trocas 4 é uma generalização do algoritmo de trocas 3.

Algorithm 4 Algoritmo de Troca Ponderada

1: L_{usd}: Lista de tuplas (id, tipo, μ_{id}, ψ_{id}) ▷ Em ordem cresc. de id
2: L_{sec}: Lista de tuplas (id, tipo, $\overline{M}_{id}$, $\sigma_{M_{id}}$) ▷ Em ordem cresc. de id
3: P: Porcentagem de instâncias para serem trocadas ▷ Em valores de 0 a 1
4: w_{usd}: Peso da medida de desempenho por custo
5: w_{sec}: Peso da medida de desempenho somente
6: **procedure** TROCA DE INSTÂNCIA($L_{usd}, L_{sec}, P, w_{usd}, w_{sec}$)
7: $\mu_{max} = 0$
8: **for** $inst$ in L_{usd} **do** ▷ Encontra μ_{max} - melhor custo
9: **if** $L_{usd}.\mu > \mu_{max}$ **then**
10: $\mu_{max} = L_{usd}.\mu$
11: $\overline{M}_{max} = 0$
12: **for** $inst$ in L_{sec} **do** ▷ Encontra $\overline{M}_{max}$ - melhor desempenho
13: **if** $L_{sec}.\overline{M} > \overline{M}_{max}$ **then**
14: $\overline{M}_{max} = L_{sec}.\overline{M}$
15: $L_{final} = \{\}$ ▷ Lista de tuplas (id, tipo, η_{id}, ϕ_{id})
16: $i = 1$; $lim = L_{sec}$
17: **for** $i \leq len(L)$ **do** ▷ Cria nova lista com as medidas η e ϕ
18: $inst.id = L_{usd}[i].id$; $inst.tipo = L_{usd}[i].tipo$
19: $inst.\eta = w_{usd} * \frac{L_{usd}[i].\mu}{\mu_{max}} + w_{sec} * \frac{L_{sec}[i].\overline{M}}{\overline{M}_{max}}$
20: $inst.\phi = w_{usd} * \frac{L_{usd}[i].\psi}{\mu_{max}} + w_{sec} * \frac{L_{sec}[i].\sigma_M}{\overline{M}_{max}}$
21: L_{final}.insert(inst)
22: L_{final}.sort(η) ▷ Ordena lista pela medida ponderada eta
23: $i = 1$
24: $lim = len(L_{final}) * P$
25: **for** $i \leq lim$ **do** ▷ Cria novas instâncias
26: **if** $L_{final}[i].tipo \neq L_{final}[len(L)].tipo$ **then**
27: **if** $L_{final}[i].\eta + L_{final}[i].\phi < L_{final}[len(L)].\eta - L_{final}[len(L)].\phi$ **then**
28: $z \leftarrow$ zona de disponibilidade de menor custo para $L_{final}[len(L)].tipo$
29: crie uma instância nova do tipo $L_{final}[len(L)].tipo$ em z
30: encerre a instância $L_{final}[i].id$

No Algoritmo 4 temos a realização da normalização e do cálculo das novas variáveis de medida de desempenho. Essas variáveis são utilizadas no final para selecionar as melhores instâncias, convergindo para um tipo ou para instâncias que apresentem resultado semelhante (isto é, com uma diferença inferior ao seu desvio padrão).

2.5 Implementação e Experimentação

Assim como mencionado na Seção 2.1 implementamos os algoritmos em *python 3.6* e utilizamos o serviço *CloudWatch* para armazenar as medidas de desempenho e o serviço *Lambda* para executar os algoritmos.

O programa foi instrumentado para medir o desempenho da aplicação NCRS em interpolações por segundo. *Scripts* em *bash* e *python* foram utilizados para extrair informações dos *logs* do programa durante sua execução enviando para o *CloudWatch* as informações necessárias.

Além disso, na Seção 2.3.1 generalizamos o conceito de intervalos de tempos. Como esses intervalos afetam o resultado final testamos diferentes períodos de tempo para a chamada das funções, especificamente testamos um minuto, três minutos e cinco minutos.

Não somente isso, mas os algoritmos 3 e 4 apresentam a variável de porcentagem de instâncias a serem trocadas a cada chamada. Escolhemos 20%, 50% e 100% para nossos testes.

Nossa implementação dos algoritmos considera apenas as máquinas virtuais que já reportaram resultados de desempenho no *CloudWatch*, ou seja, instâncias que ainda não encerraram a execução de pelo menos uma tarefa não serão considerados na lista de instâncias. Isso permite que todas as instâncias executem ao menos uma tarefa e seu desempenho seja observado antes de decidir se não é uma instância desejável para o programa. Portanto quando selecionamos 100% como porcentagem de troca, não necessariamente estamos trocando todas as instâncias vivas, mas apenas as que encerraram alguma tarefa.

Por fim, consideramos um limite de dezesseis instâncias ao mesmo tempo. Uma é ocupada pelo *Job Manager* e *Committer*, enquanto as quinze restante serão nossos *Workers*, que podem ser trocados livremente.

3 Resultados

3.1 Curva Pareto

A curva Pareto da Figura 6 foi extraída do trabalho que publicamos nos anais do XIX Simpósio em Sistemas Computacionais de Alto Desempenho [17]. Esta curva foi montada utilizando as mesmas instâncias das mostradas na Tabela 1 para o mesmo programa NCRS e seu eixo de custo (eixo das ordenadas) está em escala logarítmica de base dois para melhor visualização. Como o programa foi executado somente em uma instância de cada vez, o tamanho do dado foi reduzido para cerca de 200MB para tornar o tempo de execução viável.

É perceptível na Figura 6 que as instâncias do tipo *Spot* de fato apresentam o mesmo desempenho para um custo cerca de três vezes menor. Além disso, percebemos que existe um grupo de três instâncias (c5.9xlarge, c5.18xlarge e m5.24xlarge) que configuram a fronteira Pareto, isto é, nenhuma outra instância apresenta custo e desempenho melhores a essas instâncias (considerando este programa e os preços delas no mercado *Spot* no dia da execução).

Além disso, essa figura reforça a importância da escolha de instâncias adequada para o programa a ser executado. Para este *benchmark*, a instância c5.18xlarge no mercado

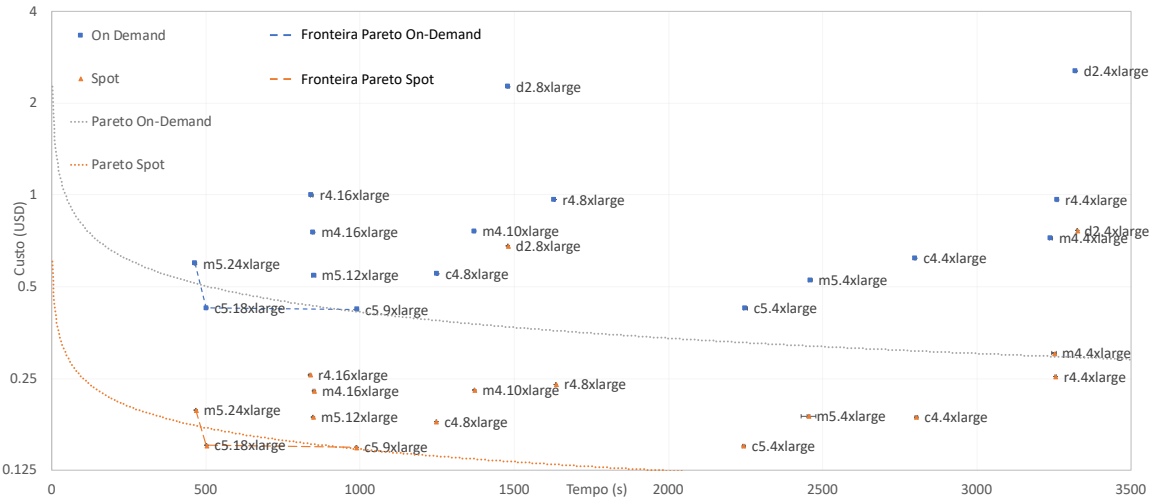


Figura 6: Curva Pareto para instâncias da Tabela 1

Spot apresenta um custo cinco vezes inferior ao da instância d2.4xlarge no mercado *Spot* (podendo chegar a dezessete vezes para a d2.4xlarge *On-Demand*), enquanto possui um desempenho sete vezes superior.

3.2 Otimização de Custo

Tendo em vista a importância da escolha de instâncias para a redução do custo de execução do programa, mostramos a motivação para a implementação de políticas de troca. Nesta subseção mostraremos os resultados experimentais da execução dos algoritmos de trocas, dadas as configurações da subseção 2.5. Como existe variação de precificação das instâncias em diferentes dias, os teste foram executados no decorrer de apenas dois dias (18 de novembro de 2018 e 19 de novembro de 2018) para comparar os métodos.

Além dos experimentos com as políticas de trocas, criamos um *baseline* que envolvia a utilização de uma instância de cada tipo dentre os mostrados na Tabela 1, sem nenhuma troca. Este *baseline* é mostrado em todos gráficos de custo, sendo utilizado para comparação entre uma execução ingênua do programa contra nossa otimização.

3.2.1 Custo com o Algoritmo 2

Tendo como base o Algoritmo 1 implementamos uma versão revisada que usa de medidas estatísticas (especificamente desvio padrão) para decidir se uma instância é de fato melhor que outra. Não executamos o Algoritmo 1, pois temos resultados já publicados no nosso trabalho nos anais do XIX Simpósio em Sistemas Computacionais de Alto Desempenho [17], nele utilizamos o mesmo conjunto de máquinas iniciais, o mesmo programa e a mesma entrada que os descritos na Seção 2. Resumidamente, no Algoritmo 1 observamos redução significativa no custo com relação a execução de apenas o conjunto inicial de máquinas, entretanto percebemos que algumas vezes as instâncias trocadas apresentavam custo por

desempenho muito semelhante às novas instâncias.

Executamos o Algoritmo 2 para três intervalos de tempo, isto é, trocamos uma máquina a cada um minuto, três minutos e cinco minutos. Na Figura 7 mostramos a evolução do custo acumulado no decorrer do tempo de execução, isto é, quanto era o total que pagaríamos (considerando somente o custo das instâncias) depois de cada minuto de execução.

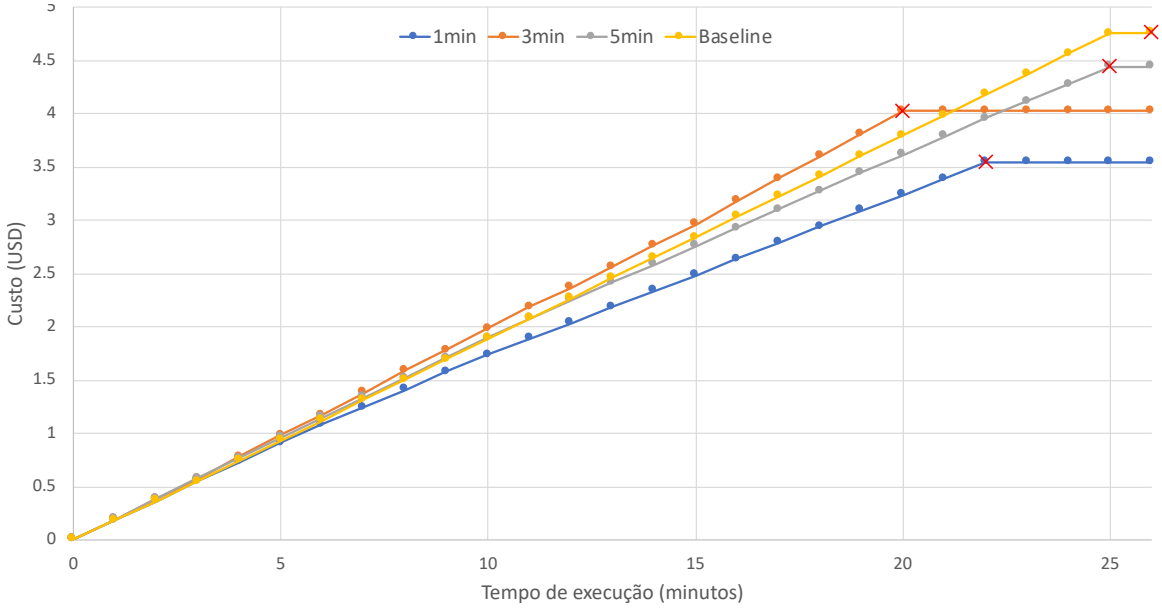
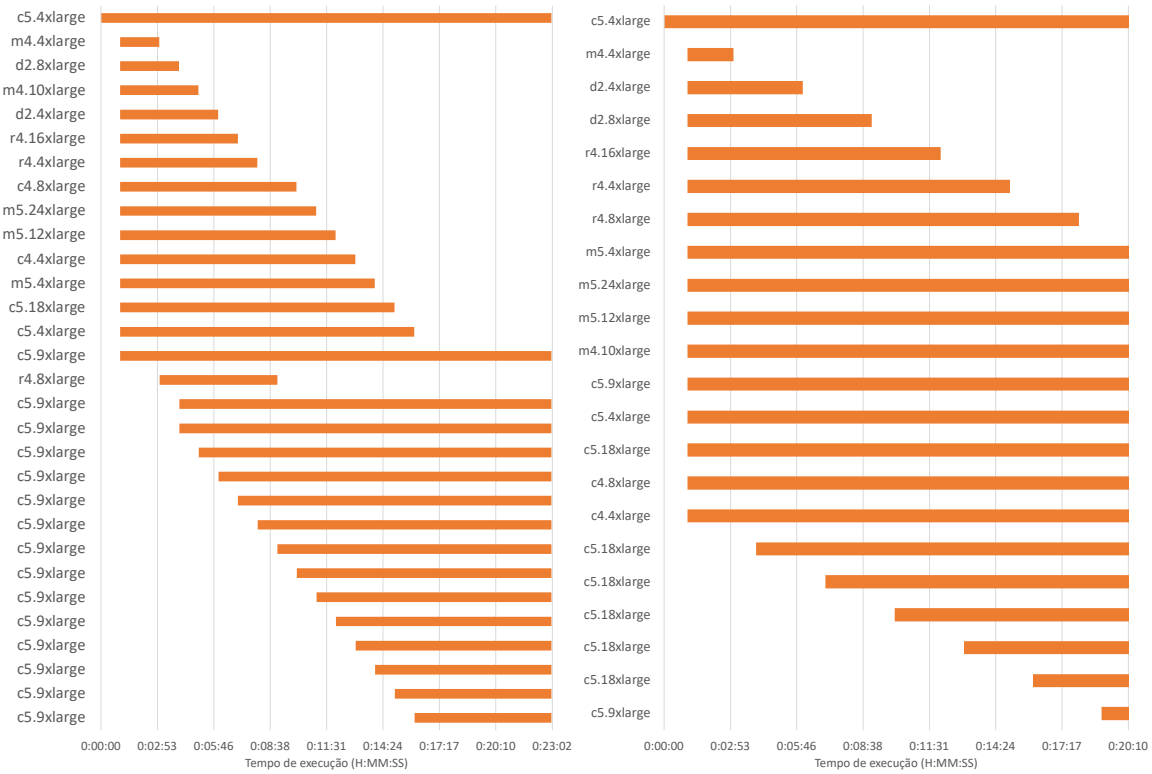


Figura 7: Evolução do custo de execução utilizando o Algoritmo 2

Na Figura 7 mostra a evolução do custo de execução no decorrer do tempo. Selecionamos cores diferentes para cada intervalo utilizado e o *baseline*, além disso marcamos com um "X"vermelho o momento em que a execução foi terminada (consequentemente as instâncias encerradas e não existe mais evolução de custo). Assim como desejado, tivemos custos inferiores ao do *baseline*, ou seja, nosso algoritmo não realizou escolhas que afetavam negativamente o custo total. Além disso obtemos ganhos em tempo também, visto que as instâncias que selecionamos além de oferecerem melhor custo também ofereceram melhor desempenho (apesar que este algoritmo não prevê que isso ocorra).

Percebemos também na Figura 7 que nenhuma política de troca apresentou o mesmo conjunto de máquinas ao final da execução. Parte do problema foi o tempo total de execução, como a execução levou menos de 26 minutos, as políticas de trocas em intervalos de três e cinco minutos não conseguiram convergir para uma única instância (como temos 15 máquinas, seriam necessários pelo menos 45 e 75 minutos de tempo de execução, respectivamente). Mas não somente isso, observamos que o ângulo da evolução de custo de com intervalo de três minutos é muito diferente com relação aos outros (sendo inclusive superior ao do *Baseline*), isso apresenta dois resultados: (i) apesar do custo maior por unidade de tempo, como o programa encerrou em menor período de tempo o custo total ainda foi inferior ao *Baseline*; (ii) o algoritmo escolheu instâncias diferentes nessas duas execuções.

Nas Figuras 8a e 8b temos o tempo de vida das instâncias no decorrer da execução



(a) Um minuto (b) Três minutos

Figura 8: Tempo de vida das instâncias para intervalos de tempo

do programa utilizando trocas a cada um minuto e a cada três minutos, respectivamente. Nestas figuras observamos os tipos de instâncias que estavam ativas durante a execução do programa, a primeira instância em ambos gráficos (c5.4xlarge) é o *Job Manager*, permanecendo ativo desde o início da execução até o último instante.

Além disso, percebe-se na Figura 8a que a instância r4.8xlarge demorou para ser inicada, isso ocorreu devido a indisponibilidade de instâncias Spot na zona de disponibilidade pedida inicialmente, sendo necessário requisitar em outra. Os algoritmos implementados não são afetados pela inicialização tardia da instância, entretanto nenhum deles oferece robustez para realizar outra requisição se existir indisponibilidade no momento da troca.

Assim como sugerido anteriormente, de fato tivemos escolhas diferentes para a melhor instância, a execução do algoritmo com intervalos de um minuto selecionou a instância c5.9xlarge como melhor opção de custo enquanto a execução com intervalos de três minutos escolheu a instância c5.18xlarge na maior parte da execução, sendo que no final a instância c5.9xlarge foi escolhida. Uma causa para escolhas de diferentes instâncias é a variação de custo em diferentes dias (ou até mesmo horas do dia). Entretanto, não houve alteração de precificação nem da instâncias c5.9xlarge nem da c5.18xlarge entre as execuções (mais especificamente, no dia 19 de novembro de 2018). Portanto, a causa para essa diferença na escolha foi oscilação no desempenho, especialmente visível na seleção de uma instância

c5.9xlarge após cinco escolhas das instâncias c5.18xlarge na Figura 8b.

Ademais, apesar da escolha diferente, o custo total de execução foi conforme o esperado: superior ao valor pago usando trocas de um em um minuto, enquanto inferior ao valor das trocas de cinco em cinco minutos. Reforçando que todos intervalos de tempo apresentaram custo de execução inferior a *Baseline*. Por fim, para esta política de troca não observamos a necessidade do desvio padrão para analisar se a troca de instância era válida. Isto é, na escolha da melhor máquina, o número de interpolações por US$ das instâncias diferiam em mais de um intervalo de desvio padrão. Porém é importante destacar que em outros testes (como o mostrado na Figura 12c) observamos que o número de interpolações por US$ das instâncias c5.18xlarge e c5.9xlarge diferiu em menos de um intervalo de desvio padrão, não sendo então trocadas uma pela outra.

3.2.2 Custo com o Algoritmo 3

Os resultados mostrados na Seção 3.2.1 já apresentaram melhoria no custo de execução. Entretanto percebemos que não houve convergência em duas das três execuções do algoritmo por falta de tempo, além disso é perceptível que o menor intervalo de tempo resultou no melhor custo, ou seja, convergir mais rapidamente para instâncias que apresentam as melhores relações de desempenho por custo resulta em melhorias de custo significativas.

Para tentar obter maior rapidez na convergência para as melhores instâncias temos a política de trocas mostrada no Algoritmo 3, em que trocamos uma porcentagem do total de instâncias de cada vez. A troca de uma porcentagem de instâncias tem como consequência redução no número de iterações para convergirmos. Para testar o algoritmo, foram realizados nove experimentos, sendo eles a combinação das variáveis de intervalos de tempo (um minuto, três minutos e cinco minutos) e agora porcentagens das instâncias serão trocadas (20%, 50%, 100%).

Na Figura 9 colocamos a evolução do custo de execução no decorrer do tempo para cada experimento. Novamente usamos cores diferentes para diferenciar os intervalos de troca, enquanto usamos marcadores diferentes para diferenciar a porcentagem trocada. O fim da execução é quando o custo no decorrer do tempo para de aumentar, isto representa o momento que todas instâncias foram desligadas. Deixamos a reta constante para facilitar a comparação de custo entre os diferentes experimentos.

Apesar da densidade de informações na Figura 9 dificultar a análise dos resultados individualmente, ainda é possível extrair algumas conclusões. A primeira conclusão é que as trocas em intervalos de um minuto e três minutos apresentaram custos semelhantes para quaisquer porcentagem trocada (em torno de US$3.5), enquanto as trocas em intervalos de cinco minutos reduziram o custo de acordo com o aumento da porcentagem de instâncias trocadas. Além disso, novamente obtemos custos inferiores ao *Baseline*, com uma redução no tempo de execução, provando novamente a efetividade do algoritmo de trocas.

Separamos a Figura 9 em três partes, mostradas na Figura 10, para melhor visualização. Cada uma das Figuras 10a, 10b e 10c apresenta a evolução do custo por tempo de execução trocando-se 20%, 50% e 100% das instâncias a cada iteração, respectivamente. Além disso comparamos para cada porcentagem a diferença no intervalo de trocas.

É perceptível que a troca de 20% das instâncias de cinco em cinco minutos apresentou

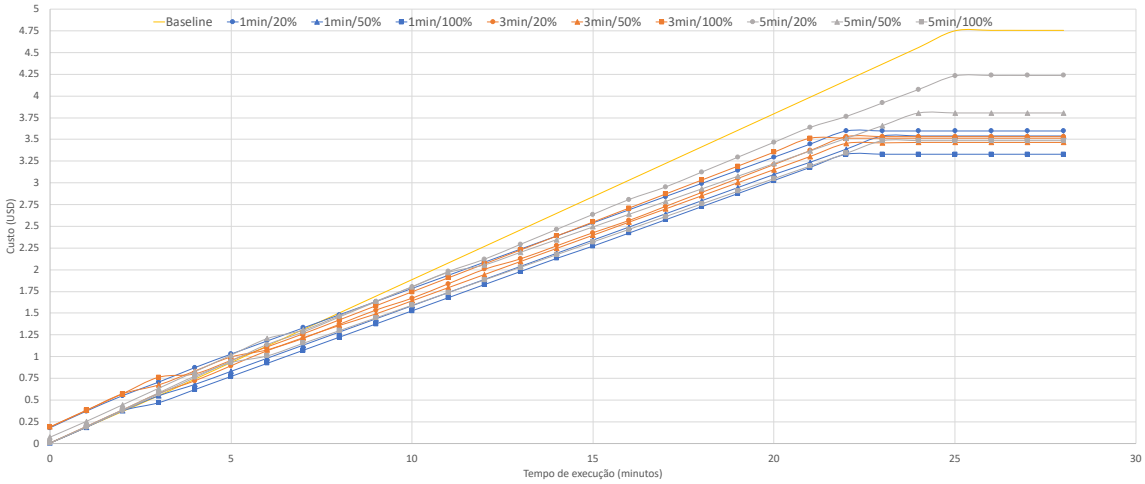
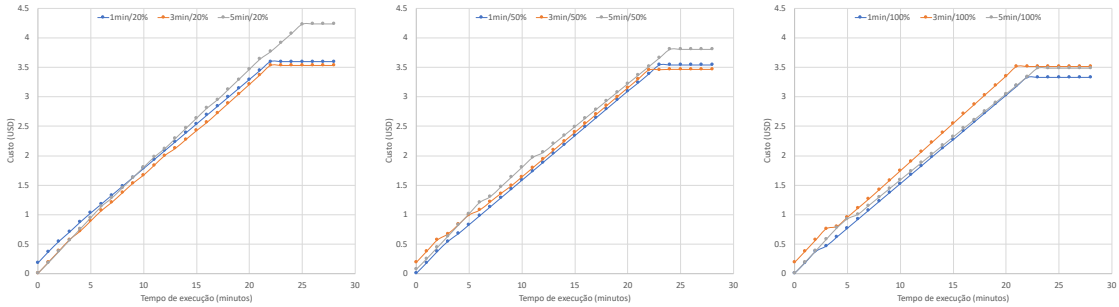


Figura 9: Evolução do custo de execução utilizando o Algoritmo 3



(a) 20% de instâncias trocadas por iteração (b) 50% de instâncias trocadas por iteração (c) 100% de instâncias trocadas por iteração

Figura 10: Custo no decorrer do tempo para o Algoritmo 3 separado em porcentagens

os piores resultados, enquanto trocar 100% das instâncias apresentou os melhores custos. Entretanto, é mais notável que no final da execução todas retas apresentaram aproximadamente o mesmo ângulo com relação ao eixo das abscissas, isto é, apresentavam a mesma derivada com relação ao tempo. O significado da derivada com relação ao tempo nestas figuras é o custo por unidade de tempo (no caso minutos). Logo, se os experimentos apresentam a mesma derivada, significa que convergimos para as mesmas instâncias. Isso implica que a diferença de custo (no geral) está relacionada com o tempo de inicialização das instâncias e o tempo para convergência e no decorrer da execução o custo acumulará igualmente.

Vamos agora analisar o tempo de vida das instâncias para alguns experimentos. Primeiramente, selecionamos a troca de 100% a cada minuto para observamos o melhor resultado de custo. Este resultado, mostrado na Figura 11, indica rapidamente que todas as instâncias foram trocadas para um único tipo (c5.9xlarge).

Nesta figura percebemos que rapidamente todas instâncias foram desligadas (exceto a

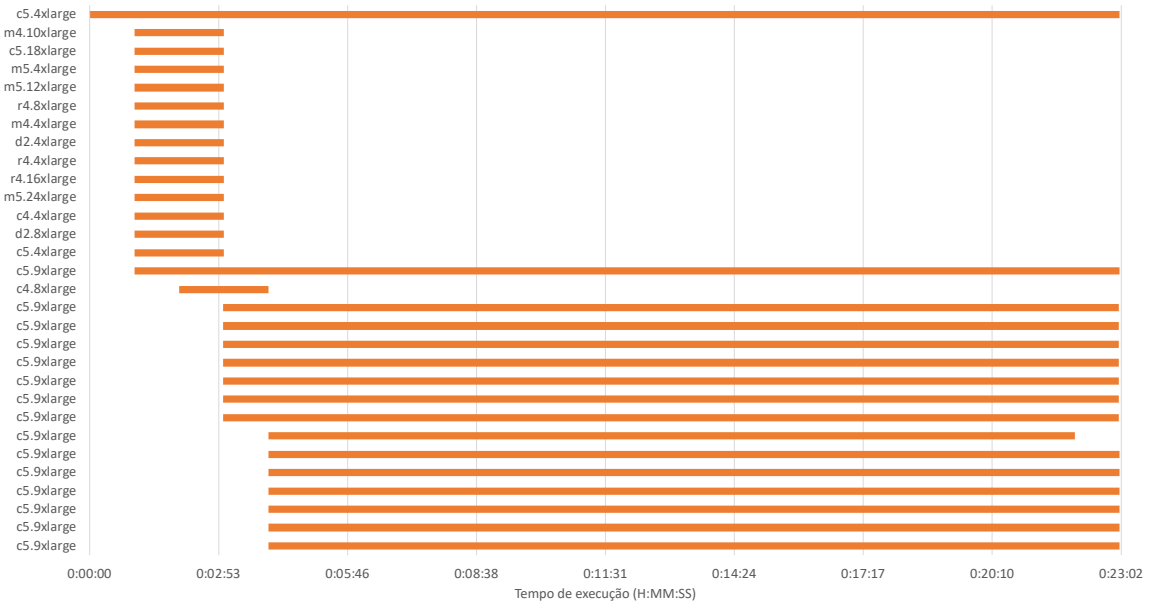


Figura 11: Tempo de vida das instâncias no decorrer da execução utilizando o Algoritmo 3 em intervalos de um minuto e 100% das instâncias trocadas

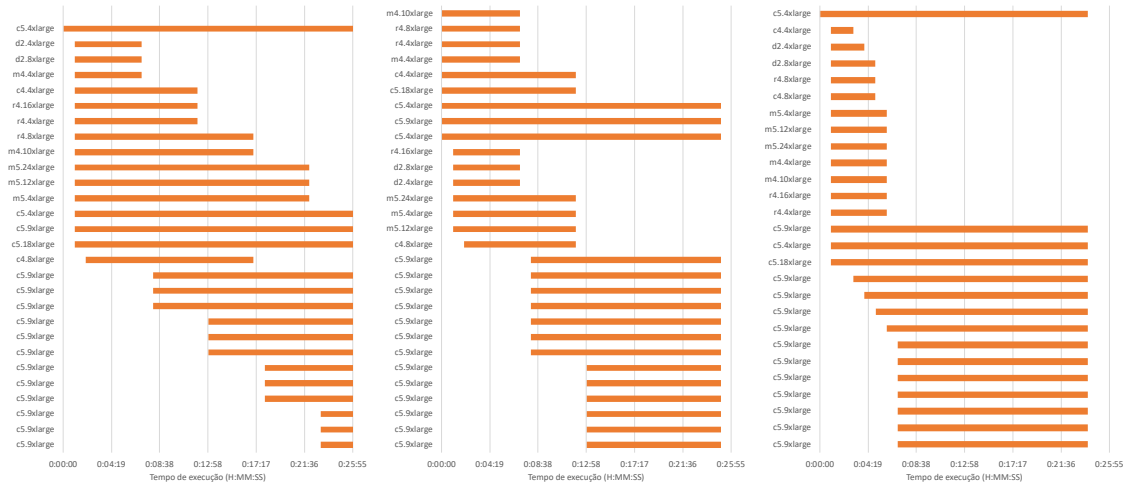
instância c5.4xlarge *Job Manager* e a própria c5.9xlarge) e substituídas por instâncias do tipo c5.9xlarge. Essa rapidez na substituição das instâncias foi o motivo para apresentarem o melhor resultado de custo.

Além disso, é perceptível que houve um pequeno atraso para serem inicializadas algumas instâncias. Esse atraso não afeta a execução do programa e pode ter sido causado simplesmente pelo tempo do próprio Algoritmo percorrer a lista de instâncias e inicializá-las. Por fim, percebemos que uma das instâncias não chegou ao final da execução, sendo encerrada prematuramente. Acreditamos que a instância tenha sido encerrada pelo provedor AWS e percebe-se que, devido a robustez a falhas nos *Workers* do SPITS, o programa ainda foi capaz de executar o último minuto normalmente gerando o resultado desejado.

Em seguida na Figura 12 mostramos a diferença do tempo de vida de instâncias para diferentes porcentagens de troca em intervalos de cinco minutos.

Observamos que em todos os casos nem todas instâncias foram substituídas pela c5.9xlarge (que repetidamente apresentou a melhor medida de desempenho por custo). As instâncias do tipo c5.4xlarge e c5.18xlarge continuaram no processamento mesmo após o tempo necessário para convergência, implicando que a medida de desempenho por custo para estas instâncias diferia em menos de um desvio padrão da instância c5.9xlarge (aplicando a equação 9).

É notável que a ordem das instâncias a serem encerradas foi semelhante nestes experimentos, mostrando que a diferença de custo realmente surgiu do intervalo mais longo entre as trocas. Além disso, é perceptível que a troca de 100% de instâncias ocorreu em passos, isto é, algumas instâncias foram encerradas antes de outras e as novas instâncias não foram colocadas ao mesmo tempo. Não conseguimos detectar o motivo para este problema.



(a) 20% de instâncias trocadas por iteração (b) 50% de instâncias trocadas por iteração (c) 100% de instâncias trocadas por iteração

Figura 12: Tempo de vida das instâncias no decorrer da execução do programa utilizando o Algoritmo 3 em intervalos de cinco minutos

Por fim, apesar dos melhores resultados surgirem das trocas de 100% das instâncias, não acreditamos que seja a melhor forma de realizar escolhas. Como visto na Figura 8b, podemos fazer escolhas erradas durante a execução do algoritmo. Isto é, uma instância que apresenta melhor desempenho por custo em determinado momento pode não ser de fato a melhor escolha (nos nossos experimentos essa escolha foi a c5.9xlarge, entretanto alguns experimentos optaram pela c5.18xlarge inicialmente). Além disso, como mencionado na Seção 2.5, somente consideramos que a instância pode ser descartada após a execução da primeira tarefa, gerando o seguinte problema:

Supondo uma situação que trocamos 100% das instâncias exceto a melhor (pois ela ainda não terminou a primeira tarefa), teremos que trocar todas instâncias novamente após o término da execução da melhor instância acarretando maior tempo de execução e custo aguardando a inicialização das novas instâncias.

Concluímos que essa política de trocas oferece uma proposta de convergência para a instância de melhor desempenho por custo em menos iterações com riscos como os supracitados. Como vimos, no final obtemos a convergência para o mesmo tipo de instância, portanto é necessário balancear o tempo de execução total com os gastos adicionais para utilizar uma solução mais segura. Ou seja, se o tempo de execução total for longo suficiente podemos utilizar uma porcentagem mais baixa de instâncias a serem trocadas (ou mesmo somente uma de cada vez) para reduzir o risco de escolhas erradas. Por outro lado, em um experimento de execução mais rápida (como o mostrado neste trabalho) pode ser válido arriscar porcentagens mais altas para convergir mais rapidamente.

3.3 Otimização de Custo/Benefício

Os algoritmos mostrados nos resultados anteriores somente otimizam o custo da aplicação, desconsiderando o desempenho. Porém como mostrado na Figura 6 a instância c5.18xlarge executou com um custo semelhante à c5.9xlarge porém em um tempo inferior. Mostraremos nesta subseção os resultados da execução do Algoritmo 4, em que ponderamos custo e desempenho, sacrificando um pouco de custo para ganhar desempenho.

3.3.1 Custo com o Algoritmo 4

Assim como nos testes da subseção 3.2.2 executamos nove experimentos, combinando diferentes intervalos de tempo com diferentes porcentagens trocadas. Similarmente, escolhemos um minuto, três minutos e cinco minutos para os intervalos, e 20%, 50% e 100% para as porcentagens. Testamos apenas com pesos iguais para custo e desempenho, portanto o algoritmo de trocas escolhe instâncias com o mesmo viés para as medidas.

Na Figura 13 mostramos a evolução do custo por tempo de execução utilizando essa política de trocas. Assim como na subseção 3.2.2 utilizamos cores diferentes para marcar os diferentes intervalos de troca e usamos marcadores diferentes para as diferentes porcentagens, enquanto o baseline não possui marcador. E novamente, o fim da execução é indicado quando o custo no decorrer do tempo não varia, representando o momento que todas instâncias foram desligadas. Deixamos a reta constante para facilitar a comparação de custo entre os diferentes experimentos.

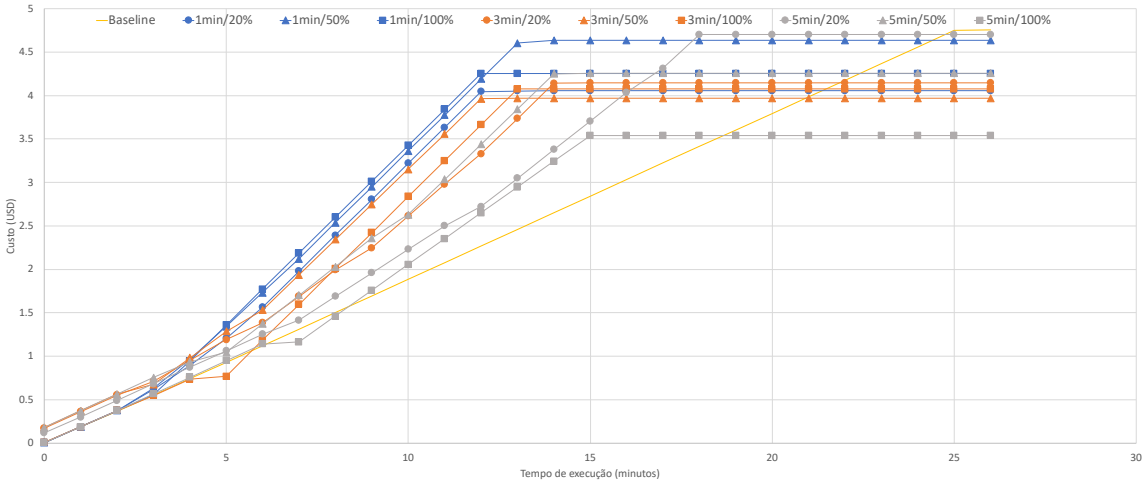
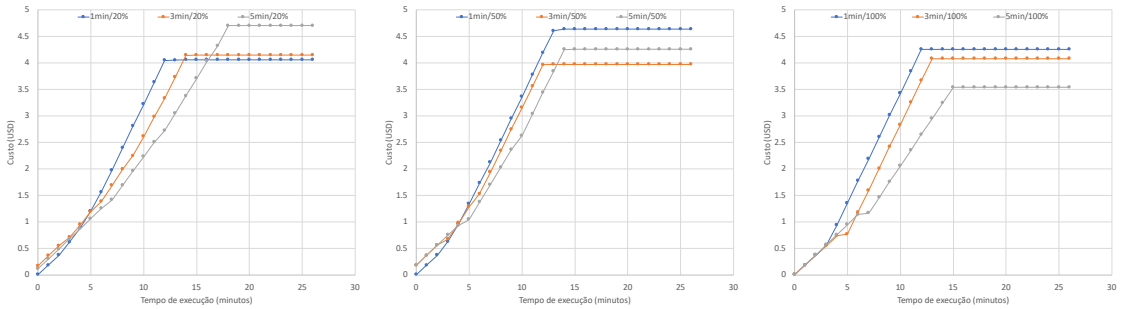


Figura 13: Evolução do custo de execução utilizando o Algoritmo 4

É perceptível na Figura 13 que as execuções acabaram consideravelmente mais rápido que o *Baseline*, enquanto ainda apresentam custos inferiores. Além disso, notamos que a maior parte das execuções custou cerca de US$4, US$0.5 a mais do que a maior parte das execuções utilizando o algoritmo 3, enquanto três execuções se destacam: as execuções com 50% de trocas a cada um minuto e 20% de trocas a cada cinco minutos se destacam



(a) 20% de instâncias trocadas por iteração (b) 50% de instâncias trocadas por iteração (c) 100% de instâncias trocadas por iteração

Figura 14: Custo no decorrer do tempo para o Algoritmo 3 separado em porcentagens

negativamente devido ao elevado custo, enquanto a execução com 100% de trocas a cada cinco minutos se destaca positivamente devido ao seu baixo custo (US$3.5).

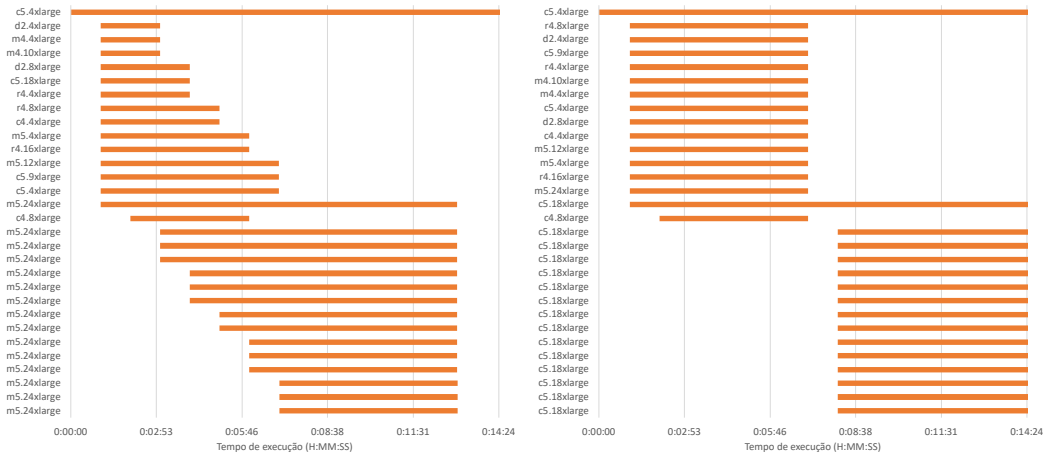
Antes de analisar o tempo de vida das instâncias para alguns experimentos, separamos a Figura 13 em três painéis, cada qual representando uma porcentagem trocada, a fim de compararmos os intervalos de troca considerando a porcentagem fixa. Esses painéis estão representados na Figura 14.

Percebemos que o algoritmo de seleção de instâncias tomou decisões que aumentaram o custo de execução para uma unidade de tempo, entretanto como essas instâncias encerram a tarefa mais rapidamente por possuírem um bom desempenho o custo total termina inferior ao *baseline*. Além disso, exceto para 20% de trocas a cada cinco minutos, todos os testes encerraram entre 10 e 15 minutos. Ressaltamos que o tempo de execução do *baseline* foram 26 minutos, portanto a execução foi executada em aproximadamente duas vezes menos tempo.

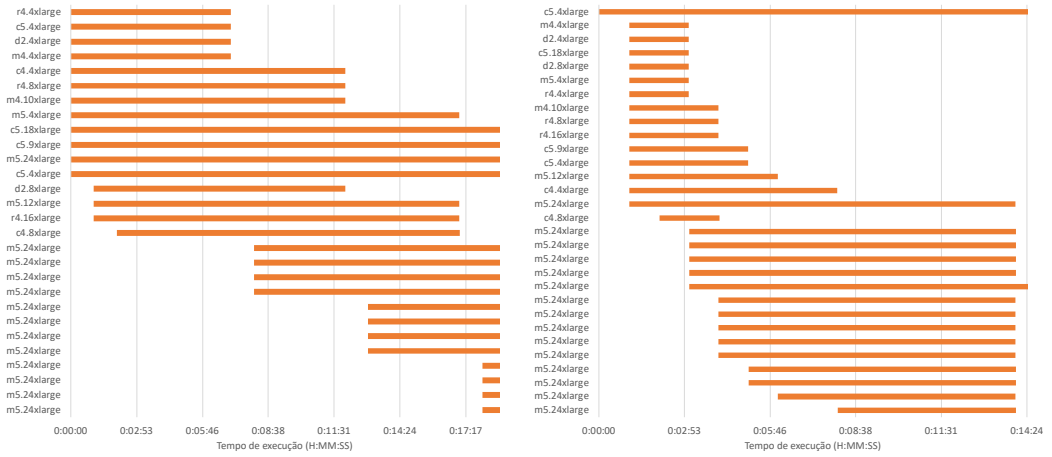
Na Figura 14c notamos que os intervalos de troca de um e três minutos escolheram o mesmo tipo de instância e manteve essa instância por toda execução enquanto a troca de cinco minutos optou por outro tipo de instância (como visto pelo coeficiente angular da reta). Por outro lado, na Figura 14a notamos que somente o intervalo de trocas de um minuto convergiu, haja vista que não tivemos tempo suficiente para convergência dos outros intervalos. Percebemos como a convergência beneficia em custo e tempo de execução o experimento.

Vamos analisar agora as instâncias que foram utilizadas no decorrer da execução de alguns experimentos. Escolhemos os três experimentos que se destacaram (50% de trocas a cada um minuto, 20% de trocas a cada cinco minutos e 100% de trocas a cada cinco minutos) e o experimento de trocas a cada minuto com 20% das instâncias trocadas para visualizar o comportamento do algoritmo (haja vista que trocamos apenas três instâncias a cada minuto) e devido ao seu tempo de execução. Colocamos todos os gráficos na Figura 15.

Na Figura 15a observamos o comportamento esperado do nosso algoritmo. A cada chamada do *script*, isto é, a cada minuto, trocamos cerca de três instâncias por instâncias que apresentam a melhor medida ponderada de desempenho mais custo por desempenho. Observamos que uma das instâncias demorou para ser inicializada (c4.8xlarge), sendo que



(a) 20% de instâncias trocadas a cada minuto (b) 100% de instâncias trocadas a cada 5 min.



(c) 20% de instâncias trocadas a cada 5 min. (d) 50% de instâncias trocadas a cada minuto

Figura 15: Tempo de vida das instâncias utilizando o Algoritmo 4

o motivo para essa demora foi a indisponibilidade desse tipo de instância na zona de disponibilidade requisitada, sendo necessário pedir a instância em outra zona. O rápido tempo de execução ocorreu devido a poucas instâncias serem terminadas durante uma troca, haja vista que quando terminamos uma instância que já estava executando não necessariamente a nova instância já começou seu processamento (é necessário aguardar tempo de *boot* e leitura de dado, por exemplo). E apesar da pouca quantidade de instâncias sendo trocadas, as novas instâncias apresentam desempenho superior às demais (assim como visto na Figura 6).

Em seguida vamos analisar a Figura 15b, haja vista que foi o experimento que apresentou o melhor custo. O motivo para esta diferença de custo está na instância selecionada pelo algoritmo. Enquanto nos outros experimentos foi escolhida a instância do tipo m5.24xlarge, neste experimento escolheu-se a instância c5.18xlarge. Como visto na curva Pareto da

Figura 6, a instância c5.18xlarge apresenta custo muito semelhante à instância c5.9xlarge porém mais desempenho. Experimentalmente observamos esse fato, haja vista que nas políticas de troca anteriores observamos a convergência para a instância c5.9xlarge com custo em torno de US$3,5, enquanto neste experimento observamos a convergência para a instância c5.18xlarge com custo em torno de US$3,5 novamente, porém com um tempo de execução menor.

Todavia, a escolha das instâncias m5.24xlarge foi mais comum nos experimentos desta subseção. Isso mostra que mesmo tendo um viés para desempenho além do custo, o algoritmo escolheu uma instância que estava na curva Pareto da Figura 6. Essa escolha mostra um custo superior à escolha de outras instâncias, porém seu tempo de execução é inferior.

Na Figura 15c observamos o motivo para o custo próximo do *baseline* no experimento trocando 20% das instâncias a cada minuto. Como nossa execução levou somente entre dezessete e dezoito minutos, efetivamente somente tivemos a oportunidade de trocar 40% das instâncias (os últimos 20% foram encerrados pouco depois de iniciados), sendo que essas trocas foram para instâncias do tipo m5.24xlarge, as quais são mais custosas que 13 das 14 outras instâncias que utilizamos.

Por fim, na Figura 15d tentamos entender o motivo para a execução destas configurações ter produzido custo mais elevado que as demais. Observando quais instâncias foram utilizadas não é suficiente para obter conclusões, haja vista que a seleção ocorreu normalmente, sendo que as instâncias foram trocadas para a m5.24xlarge assim como nas outras seleções.

Percebemos pela Figura 14b que quaisquer intervalos de tempo produziu convergência para a m5.24xlarge (dado o coeficiente angular da reta), porém a execução com um minuto de intervalo levou mais tempo que o esperado para gerar o resultado, quando comparamos a média de custo. A causa para esse tempo mais longo provavelmente foi maior tempo de inicialização ou leitura do dado, fazendo com que nosso programa trabalhe com menos instâncias por alguns minutos, já que encerramos elas antes do início de execução das novas instâncias, gerando um ou dois minutos a mais de execução que causaram essa discrepância de custo.

Por fim, mostramos que o algoritmo de troca ponderada pode trazer bons resultados de custo em conjunto com bons tempos de execução, assim como visto na Figura 15b. Porém os pesos escolhidos causaram um viés maior para tempo de execução, fazendo com que todas outras execuções optassem pela instância m5.24xlarge (que apresenta o melhor desempenho). Experimentações com outros pesos poderiam ser úteis para convergência à instância que apresenta bom balanço de custo e tempo de execução (como é o caso da c5.18xlarge na Figura 6).

3.4 Comparações de resultados

Por fim, selecionamos um resultado de cada subseção anterior para comparar as políticas de troca. Montamos o gráfico da Figura 16 comparando os custos e tempos de execução do *baseline* com o Algoritmo 2 com trocas em intervalos de um minuto, o Algoritmo 3 com trocas de 50% das instâncias a cada três minutos e o Algoritmo 4 com trocas de 100% das instâncias a cada cinco minutos. Nesta figura utilizamos cores diferentes para diferenciar cada algoritmo e o *baseline*. Mantivemos retas constantes para indicar o custo no final

da execução de cada experimento, isto é, quando o custo não varia mais indicia que o experimento foi encerrado e todas suas instâncias foram terminadas.

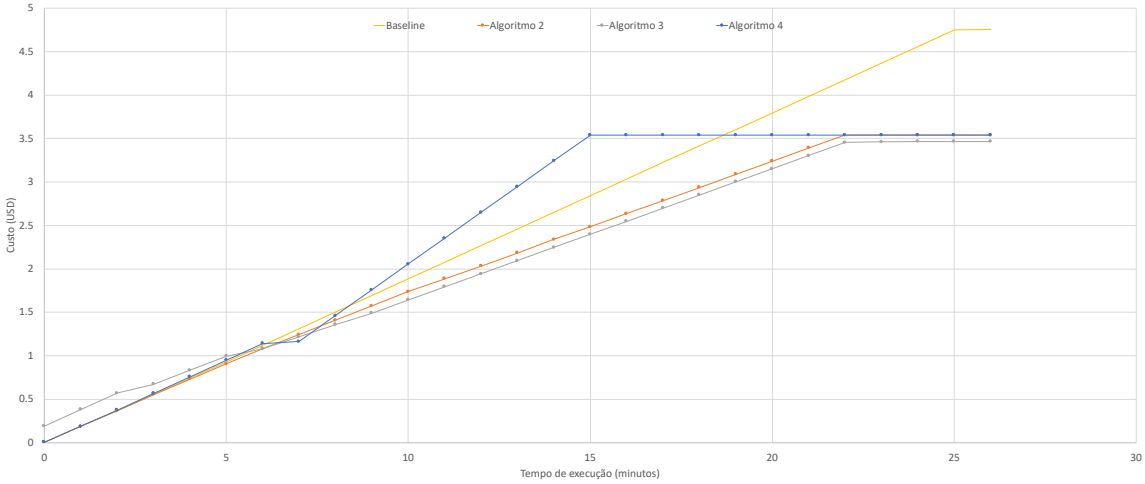


Figura 16: Comparação da evolução de custo por tempo de execução entre algoritmos

Percebemos na Figura 16 que nossas escolhas possuem custo semelhante. Não optamos pela substituição de 100% das instâncias utilizando o Algoritmo 3 devido aos problemas que podem advir dessa substituição. Porém escolhemos a substituição de 100% das instâncias a cada cinco minutos para o Algoritmo 4 para mostrar como seu custo de fato se é semelhante aos demais porém encerrando a execução em 30% menos tempo.

Além disso mostramos que mesmo a escolha mais conservadora de apenas uma instância a cada minuto oferece pequena diferença de custo total, sendo essa diferença somente causada pelo tempo a mais para convergência.

Por fim, concluímos que todos algoritmos mostrados são viáveis. A escolha de qual deles utilizar depende do risco que o usuário está disposto a correr para corrigir escolhas erradas, ou se é interessante colocar um viés de desempenho.

4 Conclusões

Neste trabalho estudamos a infraestrutura da AWS e como utilizar seus serviços para a execução de programas de alto desempenho. Dada a facilidade para adaptar nosso programa para a infraestrutura em nuvem, acreditamos que a nuvem computacional é de fato uma boa alternativa à compra de *clusters convencionais*.

Utilizamos um programa com o modelo de programação SPITS, e seu funcionamento foi também facilmente adaptável à nuvem. Além disso, o uso do SPITS permite que utilizemos instâncias do mercado *Spot* da AWS como *Workers*, permitindo que a redução de custos para a execução de nosso código, sendo que essa redução de custo pode ser de até três vezes para a mesma instância (logo provendo o mesmo desempenho).

Mostramos ainda neste trabalho a importância da seleção do tipo de instância para a aplicação a ser executada, sendo que podemos ter um desempenho mais lento e custos

maiores para seleções ruins. Para solucionar esse problema, desenvolvemos algoritmos que se utilizam de informações do desempenho do programa para tomar decisões sobre quais instâncias devemos utilizar e quais instâncias atualmente utilizadas devem ser descartadas.

Nossos resultados para os algoritmos de trocas de instâncias mostrou-se bastante promissor, em especial o resultado utilizando o Algoritmo 4 com 100% de trocas a cada cinco minutos, haja vista que ele selecionou a instância que apresentava custo quase tão bom quanto a instância de melhor custo porém um tempo de execução consideravelmente menor. Em todas situações nossos algoritmos foram capazes de apresentar custos inferiores ao de uma execução ingênua com várias instâncias de um mix inicial e em todas execuções observamos convergência para instâncias que estavam na nossa curva Pareto (c5.9xlarge e c5.18xlarge).

Verificamos também que a convergência mais veloz no Algoritmo 3 de trocas em lotes proveu menores custos, assim como visto na Figura 9 quando trocamos 100% das instâncias contra 20% das instâncias ou os resultados da Figura 7. Entretanto, é importante destacar que essa convergência em menor período de tempo traz riscos, pois em programas com desempenho variável podemos estar substituindo erroneamente instâncias por uma nova que não é a ótima (apenas mostrou melhor desempenho naquele momento). Tendo isso em vista, então é necessário balancear os riscos e ganhos ao selecionar o algoritmo, o intervalo de trocas e o tamanho do lote a ser trocado.

Concluímos que a utilização de SPITS e das políticas de troca na nuvem AWS facilitam a minimização do custo (ou da medida ponderada do Algoritmo 4) para a execução de programas de alto desempenho na nuvem computacional.

5 Generalização do Problema

Utilizamos o algoritmo para um programa que possui desempenho bem comportado desde o começo de sua execução. Este comportamento está mostrado na Figura 17, em que observamos que a variação no desempenho (representado em interpolações por segundo) é muito pequena.

Percebemos que a instância c5.9xlarge necessitou de cerca de quinze tarefas para atingir essa estabilidade, enquanto instâncias com maior poder computacional (como a c5.18xlarge e m5.24xlarge) oscilaram em torno do estável, porém essas oscilações foram pequenas. As tarefas que a instância c5.9xlarge executou antes da estabilidade representaram somente cerca de doze segundos de tempo execução, portanto não afetariam as tomadas de decisão (que levam no mínimo um minuto). Ademais, o desempenho foi inferior ao esperado para essas tarefas, pois não faziam uso de toda CPU da instância por serem muito pequenas. Vale destacar que não observamos isso nas outras instâncias pois executamos uma única tarefa com esses *Workers*, portanto, das máquinas selecionadas, somente a c5.9xlarge recebeu essas tarefas pequenas referentes ao início do dado.

Esse tipo de comportamento permite que nossos algoritmos tomem decisões logo no início da execução e que no decorrer da aplicação nossas escolhas não serão diferentes. Não somente isso, mas tomando uma decisão de trocar lotes no início da execução permite a convergência em menor tempo e, consequentemente, menor custo, assim como discutido

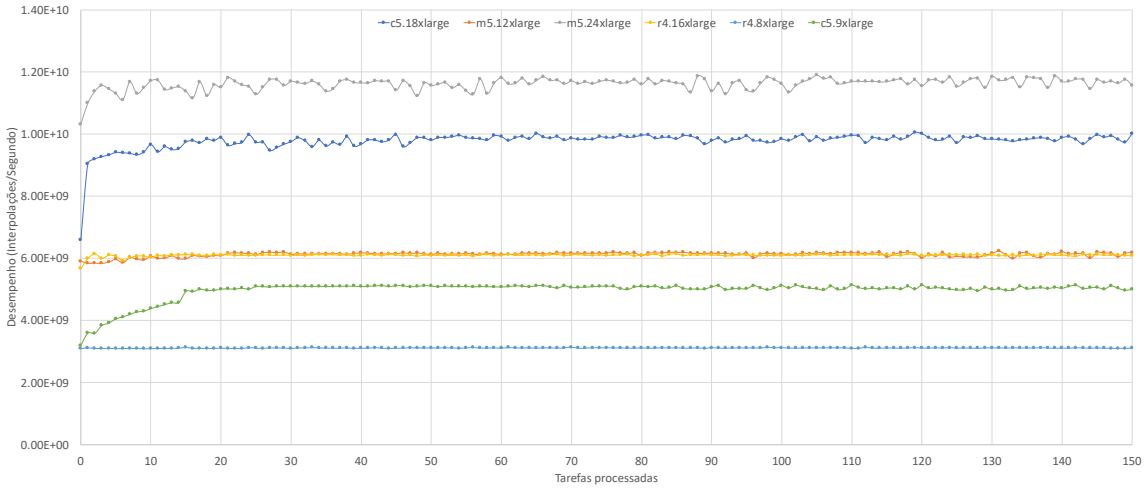


Figura 17: Comportamento do desempenho da aplicação em algumas máquinas no decorrer da execução

anteriormente. Entretanto, outras aplicações podem ter cargas de trabalho que possuem variação de desempenho com o tempo, exemplificada na Figura 18. Esses tipos de carga de trabalho possuiriam dinâmicas para seleção de instâncias diferentes das que observamos na Figura 17.

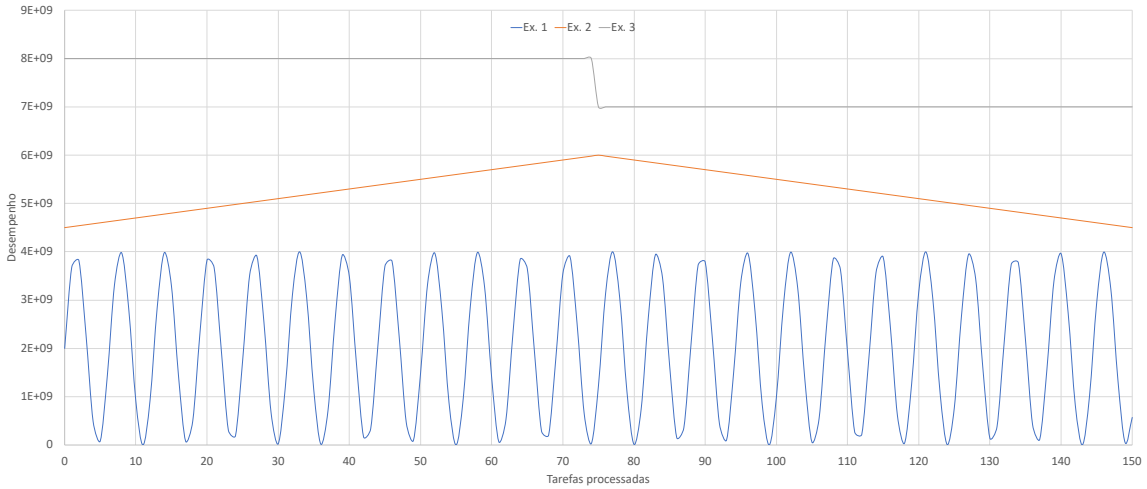


Figura 18: Exemplo de variações no desempenho no decorrer de execução de outras aplicações

Ainda não testamos nem validamos nossos algoritmos para programas com esse tipo de comportamento. É possível que nossos métodos de escolha apresentem dificuldades, tendo em vista que teríamos muitas corretivas, isto é, trocaríamos instâncias que estavam com bom desempenho e em seguida, devido à variação, essas novas instâncias seriam trocadas

novamente.

Em suma, mostramos que nosso algoritmo apresenta bons resultados para cargas bem comportadas, como as mostradas na Figura 17. O problema de otimização de custo e desempenho para aplicações com comportamento como o mostrado na Figura 18 ainda não foi testado nem resolvido neste trabalho. Porém supomos que uma troca em lotes que cresce exponencialmente o número de máquinas trocadas poderia ser utilizado para lidar com esse tipo de comportamento de desempenho.

6 Agradecimentos

Agradecemos ao laboratório *High Performance Geophysics* (HPG) e LMCAD pela infraestrutura e suporte computacional. Também agradecemos à Petrobras, à Fapesp, ao CNPq e à CAPES pelo apoio financeiro.

Referências

[1] Amazon. Amazon cloudwatch. `https://aws.amazon.com/cloudwatch/features/`, 2018. Acessado em 20/11/2018.

[2] Amazon. Amazon ebs features. `https://aws.amazon.com/ebs/features/`, 2018. Acessado em 15/11/2018.

[3] Amazon. Amazon efs features. `https://aws.amazon.com/efs/features/`, 2018. Acessado em 15/11/2018.

[4] Amazon. Amazon efs pricing. `https://aws.amazon.com/efs/pricing/`, 2018. Acessado em 15/11/2018.

[5] Amazon. Amazon lambda. `https://aws.amazon.com/lambda/features/`, 2018. Acessado em 20/11/2018.

[6] Amazon. Amazon s3 features. `https://aws.amazon.com/s3/features/`, 2018. Acessado em 15/11/2018.

[7] Amazon. Amazon s3 pricing. `https://aws.amazon.com/s3/pricing/`, 2018. Acessado em 15/11/2018.

[8] Amazon. Amazon web services (aws). `https://aws.amazon.com`, 2018. Acessado em 20/07/2018.

[9] Azure. Microsoft azure. `https://azure.microsoft.com/`, 2018. Acessado em 20/07/2018.

[10] C. Benedicto, I. L. Rodrigues, M. Tygel, M. Breternitz, and E. Borin. Harvesting the computational power of heterogeneous clusters to accelerate seismic processing. In *15th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 31 July-3 August 2017*. Brazilian Geophysical Society, Agosto 2017.

[11] E. Borin, C. Benedicto, I. L. Rodrigues, F. Pisani, M. Tygel, and M. Breternitz. Py-pits: A scalable python runtime system for the computation of partially idempotent tasks. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 7–12, Outubro 2016.

[12] E. Borin, I. L. Rodrigues, A. Novo, J. Sacramento, M. Breternitz, and M. Tygel. Efficient and fault tolerant computation of partially idempotent tasks. In *14th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 3-6 August 2015*. Brazilian Geophysical Society, Agosto 2015.

[13] Sergey Fomel and Roman Kazinnik. Non-hyperbolic common reflection surface. *Geophysical Prospecting*, 61(1):21–27, Abril 2012.

[14] Google. Google cloud. `https://cloud.google.com`, 2018. Acessado em 20/07/2018.

[15] N. Okita, T. Coimbra, and E. Borin. Análise de custo da nuvem computacional para a execução de algoritmos no processamento sísmico. In *ERAD-SP 2018, São José dos Campos, 13-15 April 2018*. SBC, Abril 2018.

[16] N. Okita, T. Coimbra, C. Rodamilans, M. Tygel, and E. Borin. Using spits to optimize the cost of high-performance geophysics processing on the cloud. In *First EAGE Workshop on High Performance Computing for Upstream in Latin America, Santander, Colombia, 21-22 September 2018*. EAGE, Setembro 2018.

[17] N. Okita, C. Rodamilans, T. Coimbra, M. Tygel, and E. Borin. Otimização automática do custo de processamento de programas spits na aws. In *Anais da Trilha Principal do XIX Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2018)*, pages 196–207. SBC, Outubro 2018.

[18] Rainer Storn and Kenneth Price. *Journal of Global Optimization*, 11(4):341–359, 1997.