Tolerância a Falhas de Workflows Científicos Executados em Nuvens Usando Checkpoints

Leonardo A. de Jesus, Daniel de Oliveira, Lúcia M. A. Drummond

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói – RJ – Brasil

{leonardoaj,danielcmo,lucia}@ic.uff.br

Abstract. *Scientific workflows are models composed of tasks, data and dependencies whose objective is to represent simulation based scientific experiments. These experiments have high demand for computational resources once they involve a number of different software processing a considerable volume of data. Thus, the use of High Performance Computing techniques provides the necessary support to the execution of such experiments and improvement of the overall delivered quality of service. Scientific Workflow Management Systems (SWfMS) are necessary in order to manage this whole range of resources. However, once High Performance Computing environments involve a number of diverse resource working in parallel, the likelihood of experiencing failures rises. Thus, SWfMS must be able to work in the presence of such faults. This work aims at implementing fault tolerance techniques in such system, improving its resiliency and reducing the workflow's makespan and incurred execution costs.*

Resumo. *Workflows científicos são modelos compostos por tarefas, dados e dependências cujo objetivo é representar experimentos científicos baseados em simulações. Estes experimentos tem alta demanda por recursos computacionais uma vez que envolvem o processamento de um grande volume de dados por diversos softwares diferentes. Assim, a utilização de técnicas de Computação de Alto Desempenho na implementação de workflows científicos fornece o apoio necessário à realização desses experimentos com maior qualidade de serviço. Para gerenciar todo este processo são necessários Sistemas de Gerência de Workflows Científicos (SGWfC). Entretanto, uma vez que ambientes de Computação de Alto Desempenho envolvem um grande número de variados recursos trabalhando em paralelo, aumenta-se a probabilidade de ocorrência de falhas em algum destes. Portanto, os SGWfC precisam ser tolerantes a tais falhas. Este trabalho busca implementar técnicas de tolerância a falhas em tais sistemas de forma a aumentar a sua resiliência e, consequentemente, diminuir o tempo total de execução e os custos envolvidos.*

1. Introdução

Um *workflow* científico é um modelo por meio do qual é possível definir uma sequência de atividades (invocações de programas e/ou serviços) e a dependência de dados entre as mesmas de forma a representar uma simulação científica [Taylor et al. 2014]. O uso de *workflows* vem crescendo nas últimas décadas em diversas áreas como a biologia, a química e astronomia. Em cada uma delas, a aplicação de *workflows* visa apoiar os

pesquisadores facilitando a obtenção de resultados que, anteriormente, haveriam de ser produzidos de forma manual, demandando uma grande quantidade de recursos humanos e tempo de pesquisa. Os *workflows* são modelados, executados e monitorados pelos Sistemas de Gerência de *Workflows* Científicos (SGWfC). Muitos desses *workflows* científicos são de larga-escala, *i.e.* consomem e produzem um grande volume de dados e comumente são executados dezenas de vezes até se confirmar ou refutar a hipótese científica associada ao experimento. Devido ao volume de dados e a demanda por processamento, esses *workflows* necessitam de ambientes de processamento de alto desempenho, como *clusters* ou *grids*, e técnicas de paralelismo para executar em tempo hábil. Entretanto, as Nuvens Computacionais têm se mostrado mais propícias nos últimos anos para executar os *workflows* de larga-escala [Hoffa et al. 2008].

As nuvens de computadores oferecem os mais variados recursos computacionais, desde máquinas virtuais (*i.e.*VMs) a armazenamento de Terabytes. Além disso, os ambientes de nuvem oferecem capacidades elásticas, *i.e.* o usuário pode solicitar mais recursos ou abrir mão de certos recursos sob demanda e o mesmo só paga pelo tempo em que realmente utilizou o serviço. Entretanto, executar um *workflow* na nuvem traz uma série de desafios associados.

Como um determinado experimento pode envolver diversas execuções de *workflows* e cada uma delas pode durar por horas ou até mesmo dias, existe uma grande chance de VMs falharem no ambiente de nuvem. De fato, vários trabalhos discutem que a probabilidade de ocorrência de uma falha cresce conforme aumenta o número de VMs utilizadas no processamento [Jackson et al. 2010]. Além disso, mesmo que uma VM não falhe efetivamente, ela está sujeita a variações de desempenho devido a procedimentos como *live migration* ou ao esgotamento de uma cota de processamento, por exemplo. Assim, torna-se fundamental que o SGWfC responsável por gerenciar a execução dos *workflows* em nuvens seja capaz de detectar e recuperar falhas. Entretanto, para que um SGWfC seja tolerante a falhas em um ambiente de nuvem com dezenas ou milhares de VMs, um *overhead* adicional é inserido nesse cenário. Passa a ser necessário não só a gerência da execução do *workflow* (*i.e.* escalonamento de atividades nas VMs), mas também de todo o ambiente no qual o mesmo é executado, buscando ainda minimizar custos (de tempo e financeiros) e maximizar a qualidade do resultado final. É importante ressaltar que esse tipo de mecanismo não é opcional, uma vez que, na nuvem, a ocorrência de falhas não mais pode ser tratada como uma exceção.

Embora diversas técnicas de tolerância a falhas (TTF) venham sendo estudadas nos últimos anos, poucos desses trabalhos focam na tolerância a falhas de *workflows* em nuvens de computadores. Ainda, os trabalhos que focam em *workflows* em nuvem consideram o seu uso apenas para fornecer recursos extras para reexecuções, e não para realizar uma recuperação da atividade a partir do momento em que ocorreu a falha. Também, poucos destes trabalhos se propõem a investigar a utilização de mecanismos de *checkpoint/restart* (C/R) como forma de recuperação de falhas em nuvens. Como a execução de *workflows* científicos envolve a utilização de programas provenientes de diversas fontes e que são caixas pretas, é impossível para o cientista inserir *checkpoints* no código original da aplicação. Logo, o controle dos *checkpoints* deve ficar a cargo do SGWfC e não dos programas que são invocados nas atividades.

Assim, este trabalho tem por objetivo propor técnicas de detecção e tolerância a

falhas na execução de *workflows* científicos executados em nuvems computacionais, utilizando *Checkpoint/Restart* a nível de sistema operacional, como forma de mitigar os efeitos da ocorrência de falhas nos diferentes recursos em execução. Conforme será mostrado em maiores detalhes nas Seções 3 e 4, a aplicação desta técnica possibilitou a utilização de C/R sem necessidade de alteração nos programas usados nos *workflows* e isto resultou em ganho de até 32,25% de desempenho nos testes realizados na presença de falhas, em termos tanto de tempo de execução quanto de custos decorrentes. Em casos em que não ocorrem falhas, o uso da abordagem proposta insere um *overhead* de cerca de 4% no tempo total de execução.

O restante do trabalho está organizado da seguinte maneira: a Seção 2 discute os trabalhos já realizados com o objetivo de prover tolerância a falhas na execução de *workflows* científicos, a Seção 3 explica a abordagem proposta pelo presente trabalho, a Seção 4 traz resultados experimentais e a Seção 5 conclui o trabalho e apresenta os problemas em aberto.

2. Trabalhos Relacionados

Conforme mencionado na seção anterior, a tolerância a falhas é uma questão chave na execução de *workflows* científicos em nuvens computacionais. Desta forma, diversos trabalhos na área foram dedicados à investigação e apresentação de soluções para este tópico. [Yu e Buyya 2005] fornece uma taxonomia através da qual é possível classificar as diversas TTFs propostas de acordo com o nível no qual são empregadas: *Workflow Level* ou *Task Level*. Dentre as TTF a nível de tarefa temos as seguintes: *Retry*, *Alternate Resource*, C/R e Replicação. Já dentre as de nível de *workflow* temos *Alternate Task*, *Redundancy*, *User-Defined Exception Handling* e *Rescue Workflow*.

De fato, grande parte dos SGWfC atuais se baseia em alguma destas técnicas, sendo *Retry* e Replicação as TTFs mais comumente implementadas, como em [Taylor et al. 2007], [Fabra 2013], [Fahringer et al. 2005] e [Baude et al. 2005]. Apesar da técnica de C/R também ser relativamente comum, é normalmente aplicada a nível de *workflow* [Zhang et al. 2009], [Hoheisel 2006]. Nesta forma de *checkpointing*, o SGWfC armazena as informações a respeito do andamento do processamento dos *jobs*. Caso o *workflow* precise ser reiniciado, é possível que se recuperem as informações já processadas com sucesso. Porém, esta técnica não é capaz de recuperar o trabalho parcialmente realizado por um *job* que falhou. Caso se trate de um *job* de longa duração, esta perda pode vir a atrasar o processamento de todo o restante do *workflow*. Isto acarreta também um maior custo na execução, caso se utilize Computação em Nuvem.

Há ainda o caso de SGWfC que suportam *checkpoints* a nível de *job* caso sejam incluídas bibliotecas próprias para tal fim nos códigos-fonte dos *softwares* em questão como [Elmroth et al. 2007] e [von Laszewski e Hategan 2005], por exemplo. Esta solução não é viável nos casos em que o *workflow* compreende a execução de programas caixas pretas, fornecidos por terceiros.

Assim, este trabalho propõe uma extensão a um SGWfC de forma a monitorar o estado das máquinas e incluir um módulo gerenciador de *checkpoints*, e se diferencia dos demais propostos ao abordar a questão do *checkpoint* de *jobs* sem que haja a necessidade de alteração de *softwares* pré-existentes. Esta solução de C/R é, de fato, empregável em quase qualquer aplicação, salvo casos explicitados em [CRIU 2016]. E é justamente este

fato que a torna tão apropriada para a utilização dentro do contexto dos SGWfC, visto que o mesmo gerencia execuções de *softwares* diversos.

3. Abordagem

A tolerância a falhas compreende duas etapas principais: (i) detecção e (ii) correção das falhas. Na etapa de deteção, algum mecanismo deve ser utilizado para que se tenha ciência se determinada execução foi realizada com sucesso ou se algum problema ocorreu durante a mesma. Nesta etapa, é importante que sejam coletadas informações a respeito das falhas ocorridas, de forma a se decidir pela ação a ser tomada na etapa seguinte, de correção. A etapa de correção tem como função a aplicação de técnicas que sejam capazes de reparar erros nas execuções de forma que o *workflow* científico possa ser processado com sucesso.

3.1. Detecção de Falhas

Esta subseção tem por objetivo examinar algumas técnicas elaboradas neste trabalho que visam a detecção de falhas em execuções de workflows científicos em nuvens computacionais. Para fins de simplicidade, a Tabela 1 traz um resumo dos termos e significados de variáveis que serão utilizados nos algoritmos desta seção e das subsequentes.

Para a tarefa de detecção, foi elaborado um algoritmo proativo chamado FTAdaptation (Algoritmo 1). Este algoritmo busca sanar possíveis fontes de falhas antes que estas ocorram efetivamente em execuções de *jobs*. Ele é composto por três métodos principais: o primeiro, chamado *VerifyClusterHealth*, é responsável por consultar o provedor de nuvem e coletar informações a respeito de cada uma das VMs instanciadas. Neste trabalho, especificamente, foi utilizada a API fornecida pela Amazon [Amazon 2016], que provê informações a respeito da conectividade, do estado das VMs (em execução, inicializando ou parada, por exemplo), dos seus *status* (ok ou com problemas), etc. Estas informações são armazenados em uma base de dados relacional do SGWfC.

Os outros dois métodos são *SelectBadMachineId* e *SelectBadMachineType*. Eles tem a mesma funcionalidade, se diferenciando somente no que diz respeito à informação retornada, no caso o ID e o tipo da máquina (t2.small, m4.large, etc), respectivamente. Eles consultam as informações providas pelo método *VerifyClusterHealth* juntamente com informações relativas ao histórico de execuções das ativações. Por meio deste histórico é possível que se verifique se falhas ocorrem com uma frequência maior que dado limite em alguma das máquinas pertencentes ao *cluster* virtual. Caso esta situação seja observada, infere-se que a máquina em questão está sofrendo algum tipo de pro-

VM	*Virtual Machine*
TTF	Técnica de Tolerância a Falhas
FT	Fault Tolerance
C/R	*Checkpoint/Restart*
PWS	Workspace de um processo
VC	Virtual Cluster
mList	Lista de Máquinas
pid	ID do processo no Sistema Operacional
typeList	Lista de tipos de máquinas onde foram detectados problemas
badMachineIds	Lista com IDs das máquinas que foram detectadas como problemáticas

Tabela 1. Glossário de Termos

blema. Assim, o tipo e ID de tal máquina são obtidos e é possível a instanciação de uma máquina com as mesmas características, além da terminação da máquina problemática.

Algoritmo 1 FTAdaptation()

```
1:  while workflow is not finished do
2:      VerifyClusterHealth()
3:      $typeList = SelectBadMachineType()
4:      if $typeList is not empty then
5:          Stop SWfMS
6:          RequestMachines($typeList)
7:          $badMachineIds = SelectBadMachineId()
8:          RequestMachineTermination($badMachineIds)
9:          Start SWfMS
10:     Sleep($param)
```

Ao contrário da nossa abordagem proposta, a maioria dos SGWfC são reativos, ou seja, após a execução de cada uma dos *jobs*, é analisado o *Exit_Status* retornado pelo sistema operacional. A falha é detectada quando são retornados valores diferentes de 0. Ao detectar-se uma falha, simplesmente reexecutam o *job*.

3.2. Tolerância a Falhas por meio de *Checkpoints*

Nesta seção apresentamos a abordagem definida neste trabalho como forma de tornar um SGWfC tolerante a falhas. Para fins didáticos, explicaremos usando o SGWfC SciCumulus [De Oliveira et al. 2010] [SciCumulus 2016], que foi utilizado no estudo de caso, na Seção 4. O esquema apresentado na Figura 1 representa a execução de um *workflow* SciCumulus em um provedor de Computação em Nuvem utilizando 4 VMs. Em cada uma das VMs temos a execução de uma instância do SCCore, o componente responsável pela execução do *workflow* no SciCumulus. Este componente é responsável pelo escalonamento, instrumentação e execução das atividades do *workflow*. A comunicação entre as VMs é realizada por meio de mensagens MPI. A VM de *rank* 0 é responsável pela tarefa de escalonar as atividades enquanto as demais VMs aguardam requisições de execução.

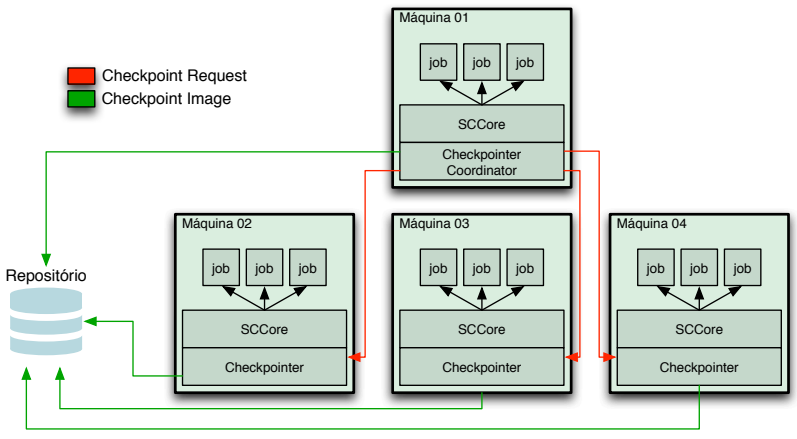


Figura 1. Arquitetura do Checkpointer Distribuído

Para implementar C/R nesse SGWfC, foi adicionada uma funcionalidade à VM Mestre ($SCCore_0$). Ela é descrita pelo Algoritmo 2, sendo a responsável pela verificação de quais *jobs* são elegíveis a um *checkpoint*. Esta verificação é realizada levando-se em consideração o tempo de execução das tarefas, que é consultado na base de dados via SQL - assume-se que há disponível uma base de dados relacional contendo o registro completo do histórico de execução, como é o caso do SciCumulus. No momento em que um job_i tem seu tempo desde o último *checkpoint* gravado $t_i > X$, onde X é um parâmetro de entrada, uma requisição de *checkpoint* é enviada ao SCCore da VM onde o job_i está sendo executado. Em cada uma das VMs foi adicionada também a funcionalidade *CheckpointListener*, representada pelo Algoritmo 3. Este algoritmo aguarda requisições de *checkpoints* e executa o Algoritmo 4.

Esta requisição contém o *workspace* - um diretório no sistema operacional - destinado ao *job* em questão. Uma vez que cada *job* está associado a um *workspace* único, é trivial a obtenção do PID deste *job* através de comandos do sistema operacional. Tal processo é então pausado, sua imagem é gravada em disco por meio da ferramenta CRIU [CRIU 2016], arquivos contendo dados de entrada ou saída parciais são copiados e incluídos juntamente à imagem do processo e este conjunto é enviado a um repositório seguro, escalável e acessível por todas as VMs. Por fim, o processo é reiniciado.

Algoritmo 2 CheckpointCoordinator()

```
1: while workflow is not finished do
2:     $mList = GetCheckpointInformation()
3:     for each $m in $mList do
4:         Send($m.PWS, $m.rank)
5:     Sleep($param)
```

Algoritmo 3 CheckpointListener()

```
1: while workflow is not finished do
2:     Receive($msg)
3:     Call Checkpoint($msg.PWS)
```

Algoritmo 4 Checkpoint($\$PWS$)

```
1: $pid = pgrep($PWS) * Consulta o PID do processo baseado em seu workspace *
2: Dump($pid,$dest) * Grava a imagem do processo e o deixa em estado parado *
3: Compress($PWS)
4: kill -SIGCONT $pid
5: Compress($dest)
6: Move($compressedFiles,$repository)
7: Delete($oldCheckpoint)
```

Na eventualidade da ocorrência de uma falha em um *job* qualquer (que pode ser verificada através do *Exit_Status* do *job* a nível de SO), o escalonador do SGWfC realiza uma tentativa de reexecução da ativação e, possivelmente, haverá um *checkpoint* gravado para a mesma. Neste caso, em vez de realizar a execução da ativação a partir do ponto

inicial, é executado o Algoritmo 5, que copia e descompacta os dados referentes ao *checkpoint* do repositório para o *workspace* da ativação e realiza a restauração. A utilização de C/R é possível também em um caso onde uma ativação esteja sendo executada há muito tempo em uma determinada VM. Ao detectar tal situação, o escalonador pode decidir por migrá-la para uma VM mais potente disponível e, ao invés de executá-la desde o início, executa-a a partir do último *checkpoint* gravado. A Figura 2 apresenta a base de dados que apoia a execução dos *workflows* no SGWfC SciCumulus, e a extensão realizada na mesma de forma a dar apoio a implementação das técnicas de tolerância a falhas descritas neste trabalho, além das técnicas de detecção de falhas tratadas na subseção anterior.

Algoritmo 5 Restart($\$Path$)

1: *$ckpt = Copy($Path)*
2: *Extract($ckpt,$dest)*
3: *Restart($dest)*

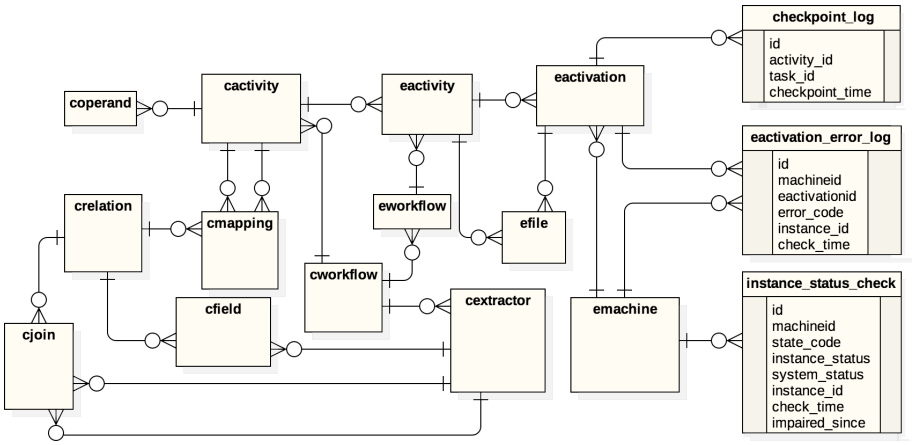


Figura 2. BD do SciCumulus com FT

Foram adicionadas as seguintes tabelas: (i) *checkpoint_log*, (ii) *eactivation_error_log* e (iii) *instance_status_check*. Na tabela *checkpoint_log* são armazenados dados relativos aos *checkpoints* realizados pelo SGWfC. São gravados o *timestamp* no momento da requisição de gravação do *checkpoint* e a referência à *eactivation* (tabela que armazena os registros referentes aos *jobs* que devem ser executados e seus respectivos dados de entrada) que deve ter seu processo gravado. Já na tabela *eactivation_error_log*, são inseridas informações acerca das falhas nas execuções das *eactivations*. Cada vez que a execução do processo relativo a uma *eactivation* retorna qualquer valor diferente de "sucesso", é inserida uma tupla nesta tabela com referências à *eactivation* em questão, à VM na qual o processamento com falha ocorreu e ao ID da instância desta VM, além do código e do *timestamp* do erro. Por fim, foi adicionada também a tabela *instance_status_check*, onde são gravados dados coletados a partir do provedor de Nuvem.

Os Algoritmos 4 e 5 (Checkpoint e Restart) se apoiam em uma solução externa de C/R para seu funcionamento. Especificamente, os comandos *Dump* e *Restart* destes algoritmos tratam de chamadas realizadas a tal solução. Neste trabalho, a solução adotada foi o CRIU [CRIU 2016]. Esta decisão foi baseada em algumas vantagens desta ferramenta

em relação a outras soluções semelhantes (BLCR [Hargrove e Duell 2006], por exemplo), como: (i) utilização sem necessidade de carregamento de bibliotecas antes do início da execução do processo, (ii) possibilidade de execução em programas sem necessidade de modificações ou qualquer tipo de preparação nos mesmos, e (iii) gravação do estado de arquivos abertos. Estas características são fundamentais no contexto da execução de *workflows* científicos, uma vez que estes, muitas vezes, tratam a execução de programas fornecidos por terceiros, caixa-preta e estaticamente ligados. Maiores informações a respeito do funcionamento desta solução podem ser obtidas em [CRIU 2016].

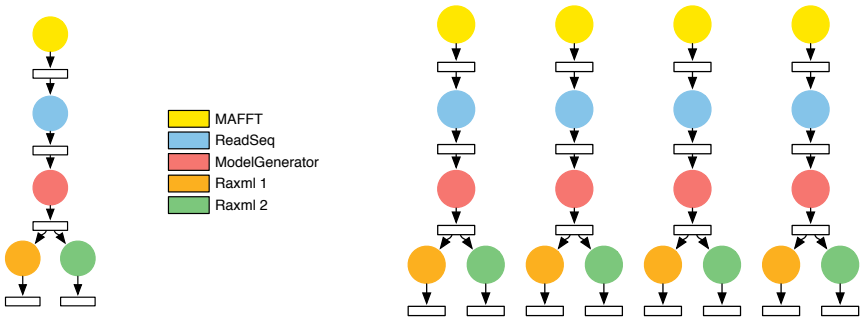
4. Resultados Experimentais

Foram realizados dois experimentos como forma de se avaliar o desempenho da abordagem proposta neste trabalho. Nestes experimentos, foram modeladas instâncias do *workflow* SciPhy [Ocaña et al. 2011], que podem ser vistas na Figura 3, com 1, 2 e 4 entradas. Estas instâncias foram executadas em VMs Amazon EC2, do tipo m4.large, rodando o sistema operacional Ubuntu 14.04.

O primeiro experimento teve como objetivo mensurar o *overhead* introduzido na execução dos *workflows* pela solução de C/R. Foram realizadas execuções de um *pipeline* do SciPhy no ambiente de teste, utilizando-se 1, 2 e 4 VMs. Desta forma, a cada VM fica atribuída a execução de um *pipeline* (equivalente a uma entrada). Como o primeiro experimento foi realizado com apenas uma VM, pode-se considerar que não há complexidade introduzida pela solução de C/R com envio e recebimento de mensagens e monitoramento dos tempos de execução dos processos nas diferentes VMs.

Foram gravados *checkpoints* das tarefas a cada minuto. Por se tratar de um exemplo com entradas pequenas, a única atividade a sofrer a gravação de seu estado foi a terceira atividade (ModelGenerator [ModelGenerator 2016]) - todas as outras foram executadas em menos de 1 minuto. Os tempos de execução foram, em média, de 4,5 minutos. É importante frisar que foi utilizado um exemplo com entradas pequenas de forma a possibilitar a realização de testes um pouco mais exaustivos. Instâncias reais deste mesmo problema podem ser executadas por dias, conforme pode ser visto em [Ocaña et al. 2011].

O experimento consistiu em 20 execuções dos *workflows* para cada configuração do *cluster* virtual. Em 10 destas execuções, foi implementado no SGWfC apenas a técnica de *Retry*, enquanto nas 10 restantes estavam implementadas as estratégias C/R e *Retry*



(a) workflow abstrato (b) execução do SciPhy com 4 entradas

Figura 3. Workflow SciPhy

(caso ocorressem falhas antes de serem gravadas imagens dos *jobs*). Entretanto, não ocorreram falhas nestas execuções. Desta forma, não há tempos adicionais decorrentes de possíveis reexecuções, ilustrando o pior cenário para a estratégia de C/R. Foram coletados os tempos de execução dos *workflows* em cada um dos cenários. As médias dos tempos de execução (em segundos) deste experimento podem ser vistos na Tabela 2.

A Figura 4(a) apresenta melhor os resultados obtidos neste experimento. Ao compararmos as médias dos tempos de execução das duas abordagens, percebemos um valor ligeiramente maior na estratégia de C/R, conforme o esperado, pois na estratégia de C/R há a necessidade de se pausar o processo de tempos em tempos. Esta diferença se mantém relativamente estável, com exceção do primeiro experimento realizado com 4 VMs, onde se observa um aumento na diferença entre o desempenho utilizando-se a estratégia C/R e a estratégia *Retry*. Entretanto, conforme pode-se verificar no segundo experimento com 4 VMs (marcado com * na Tabela 2), esta diferença de desempenho volta a ser consistente com os valores observados anteriormente. Esta divergência é explicada pela diferença entre as entradas utilizadas nos experimentos: no primeiro, temos entradas distintas enquanto no segundo são utilizadas, para os 4 *pipelines*, a mesma entrada utilizada nos experimentos com 1 VM.

Assim, comparando os resultados obtidos pelo segundo experimento com 4 VMs e o experimento com somente 1 VM, percebe-se que não há grande divergência entre os desempenhos das estratégias de C/R e *Retry*. Estes resultados indicam que a abordagem proposta é de baixo *overhead*. Ainda, a comparação entre o aumento do *overhead* para 1 VM (3,29%) e para 4 VMs (2,32%) indica que a solução é escalável, ou seja, o aumento do número de máquinas não influencia em perda de desempenho, apesar do gasto com troca de mensagens e consultas ao banco de dados.

Assim como o primeiro, o segundo experimento realizou 20 execuções sendo 10 utilizando-se *Retry* e 10 utilizando-se C/R + *Retry*. Entretanto, desta vez, o objetivo é a medição do tempo de execução dos *workflows* na presença de falhas. Como o momento em que a falha ocorre influencia na quantidade de trabalho perdida, são modeladas 4 situações: sem ocorrência de falhas, falhas ocorridas no primeiro terço da execução do *job* (início), no segundo terço (meio) e no terceiro terço (fim). Uma quinta situação (representada na tabela com um *) é modelada com falhas ocorridas no fim da execução, mas com intervalo entre *checkpoints* de 30 segundos. Os resultados deste experimento se encontram na Tabela 3.

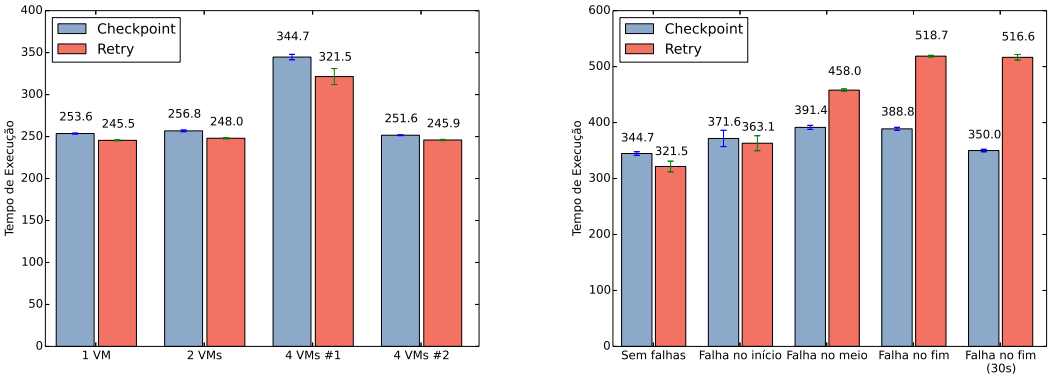
Pode-se observar que, para execuções sem falhas ou com falhas ocorridas no início do processamento, ou seja, antes da primeira gravação de *checkpoint*, a estratégia de *Retry* obtém um desempenho melhor, por motivos já discutidos no experimento anterior. En-

VMs	Retry	C/R
1	245,5s	253,6s +3,29%
2	248,0s	256,8s +3,54%
4	321,5s	344,7s +7,21%
4*	245,9s	251,6s +2,32%

Tabela 2. *Overhead* com C/R

Momento	Retry	C/R
Início	363,1s	371,6s +2,34%
Meio	458,0s	391,4s -14,54%
Fim	518,7s	388,8s -25,04%
Fim*	516,6s	350,0s -32,25%

Tabela 3. Desempenho das TTF



(a) Overhead gerado pela solução de C/R

(b) Tempo de execução na presença de falhas

Figura 4. Resultados dos Experimentos

tretanto, percebe-se uma mudança neste cenário conforme o momento de ocorrência da falha se aproxima do momento em que o processo terminaria sua execução. De fato, esta situação pode ser percebida ao se analisar a Figura 4(b). Pode-se perceber que para falhas ocorridas no meio e no fim da execução do *job*, a estratégia C/R obtém desempenho melhor que a estratégia *Retry*. Este desempenho pode ser melhorado ao se escolher um intervalo entre *checkpoints* adequado. Isto é indicado pelo resultado do segundo experimento com falhas no fim, onde o intervalo entre *checkpoints* foi reduzido em 30 segundos e isto resultou em um tempo de execução médio 30 segundos menor.

Portanto, podemos concluir que a utilização da TTF *Retry* resulta em um aumento no tempo final de execução do *workflow* conforme o momento de ocorrência da falha se aproxima do momento em que o processo terminaria sua execução. Porém, caso a TTF aplicada seja C/R, temos que este tempo permanece relativamente constante, limitando a perda ao intervalo entre *checkpoints*.

5. Conclusão e Trabalhos Futuros

Este trabalho teve por objetivo investigar diferentes técnicas de detecção e recuperação de falhas aplicadas no contexto de *workflows* científicos executados em nuvens computacionais. Foi realizado também um estudo comparativo com duas TTF, de forma a indicar a aplicabilidade desta solução em execuções reais de larga escala.

Em termos de detecção de falhas, foi proposta a utilização de dados fornecidos pelos provedores de computação em nuvem para se decidir a respeito de uma possível troca de máquinas, por exemplo. Adicionalmente, foi implementada uma solução baseada no histórico da execução dos *workflows* por meio do qual foi possível se inferir a ocorrência de problemas em VMs cujo histórico mostrasse a ocorrência de falhas acima de determinado limite percentual. Uma vez que este trabalho carece de experimentos relativos a estas funcionalidades, isto será tratado nos próximos passos da pesquisa.

Foi proposta a utilização da técnica de *Checkpointing* aliada à ferramenta CRIU [CRIU 2016] de forma a possibilitar a gravação de *checkpoints* sem que fosse necessário alterar os programas pré-existentes utilizados nas atividades dos *workflows*. Nos experimentos realizados, esta solução se mostrou de baixo custo, além de eficaz.

Foi observado que a implementação da estratégia de C/R gerou um pequeno *overhead* no tempo de execução dos *workflows*. Entretanto, ao se observar seu desempenho diante da ocorrência de falhas, é fácil concluir que os benefícios proporcionados por esta técnica fazem com que os custos sejam irrelevantes. Há de se levar em consideração que esta técnica pode não ter desempenho tão satisfatório em casos de falhas em tarefas com características diferentes. Em caso de tarefas que manipulem arquivos muito grandes, por exemplo, o tamanho das imagens de *checkpoint* geradas, bem como sua movimentação no sistema de arquivos, representa novos desafios. Assim, é necessário que sejam realizados estudos mais aprofundados que investiguem melhor esta situação.

É importante notar também que não foi realizado qualquer tipo de estudo acerca da otimização intervalo entre gravações de *checkpoints*. Alguns estudos tratam esta questão, como [Di et al. 2013] e [Young 1974]. A escolha cuidadosa deste parâmetro influi no aumento *overhead* introduzido e na diminuição da perda em caso de falhas. A utilização outras técnicas como, por exemplo, o escalonamento de tarefas *backups*, pode também oferecer melhorias aos SGWfC. É necessário que estas outras técnicas sejam investigadas de forma a se definir o melhor tipo de estratégia a ser adotada de acordo com as características do ambiente, da falha e do *job* em execução. Desta forma, a pesquisa estará abordando estes temas em uma próxima fase. Espera-se que o sucesso desta pesquisa possa fornecer melhores resultados aos cientistas, possibilitando economia e redução do tempo necessário à realização de seus experimentos.

Agradecimentos

Os autores desse artigo agradecem a CAPES, CNPq e FAPERJ por financiarem parcialmente esse trabalho.

Referências

Amazon (2016). Amazon aws. Em `http://docs.aws.amazon.com/cli/latest/reference/ec2/describe-instance-status.html`. Accessed: 2016-08-01.

Baude, F., Caromel, D., Delbé, C., e Henrio, L. (2005). A hybrid message logging-cic protocol for constrained checkpointability. Em *European Conference on Parallel Processing*, páginas 644–653. Springer.

CRIU (2016). Criu. Em `https://criu.org/Main_Page`. Accessed: 2016-05-24.

De Oliveira, D., Ogasawara, E., Baião, F., e Mattoso, M. (2010). Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. Em *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, páginas 378–385. IEEE.

Di, S., Robert, Y., Vivien, F., Kondo, D., Wang, C.-L., e Cappello, F. (2013). Optimization of cloud task processing with checkpoint-restart mechanism. Em *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, páginas 1–12. IEEE.

Elmroth, E., Hernández, F., e Tordsson, J. (2007). A light-weight grid workflow execution engine enabling client and middleware independence. Em *International Conference on Parallel Processing and Applied Mathematics*, páginas 754–761. Springer.

Fabra, J. (2013). Using cloud-based resources to improve availability and reliability in a scientific workflow execution framework.

Fahringer, T., Prodan, R., Duan, R., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.-L., Villazon, A., e Wieczorek, M. (2005). Askalon: A grid application development and computing environment. Em *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, páginas 122–131. IEEE Computer Society.

Hargrove, P. H. e Duell, J. C. (2006). Berkeley lab checkpoint/restart (blcr) for linux clusters. Em *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing.

Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B., e Good, J. (2008). On the use of cloud computing for scientific workflows. Em *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, páginas 640–645. IEEE.

Hoheisel, A. (2006). Grid workflow execution service-dynamic and interactive execution and visualization of distributed workflows. Em *Proceedings of the Cracow Grid Workshop*, volume 2, páginas 13–24. Citeseer.

Jackson, K. R., Ramakrishnan, L., Runge, K. J., e Thomas, R. C. (2010). Seeking supernovae in the clouds: A performance study. Em *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, páginas 421–429, New York, NY, USA. ACM.

ModelGenerator (2016). Modelgenerator. Em `http://mcinerneylab.com/software/modelgenerator/`. Accessed: 2016-05-24.

Ocaña, K. A., de Oliveira, D., Ogasawara, E., Dávila, A. M., Lima, A. A., e Mattoso, M. (2011). Sciphy: a cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes. Em *Brazilian Symposium on Bioinformatics*, páginas 66–70. Springer.

SciCumulus (2016). Scicumulus. Em `https://scicumulusc2.wordpress.com`. Accessed: 2016-07-24.

Taylor, I., Shields, M., Wang, I., e Harrison, A. (2007). The triana workflow environment: Architecture and applications. Em *Workflows for e-Science*, páginas 320–339. Springer.

Taylor, I. J., Deelman, E., Gannon, D. B., e Shields, M. (2014). *Workflows for e-Science: scientific workflows for grids*. Springer Publishing Company, Incorporated.

von Laszewski, G. e Hategan, M. (2005). Java cog kit karajan/gridant workflow guide. Technical report, Technical Report, Argonne National Laboratory, Argonne, IL, USA.

Young, J. W. (1974). A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531.

Yu, J. e Buyya, R. (2005). A taxonomy of scientific workflow systems for grid computing. *ACM Sigmod Record*, 34(3):44–49.

Zhang, Y., Mandal, A., Koelbel, C., e Cooper, K. (2009). Combined fault tolerance and scheduling techniques for workflow applications on computational grids. Em *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, páginas 244–251. IEEE Computer Society.