UNIVERSIDADE FEDERAL FLUMINENSE

LEONARDO ARAÚJO DE JESUS

TOLERÂNCIA A FALHAS EM WORKFLOWS CIENTÍFICOS EXECUTADOS EM NUVENS COMPUTACIONAIS

NITERÓI

2017

UNIVERSIDADE FEDERAL FLUMINENSE

LEONARDO ARAÚJO DE JESUS

TOLERÂNCIA A FALHAS EM WORKFLOWS CIENTÍFICOS EXECUTADOS EM NUVENS COMPUTACIONAIS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação.

Orientador:
LÚCIA MARIA DE ASSUMPÇÃO DRUMMOND

Co-orientador:
DANIEL CARDOSO MORAES DE OLIVEIRA

NITERÓI

2017

LEONARDO A. DE JESUS

Tolerância a Falhas em Workflows Científicos Executados em Nuvens Computacionais

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação

Aprovada em Setembro de 2017.

BANCA EXAMINADORA

Prof. D.Sc. Lúcia Maria de Assumpção Drummond - Orientadora, UFF

Prof. D.Sc. Daniel Cardoso Moraes de Oliveira - Co-orientador, UFF

Prof. D.Sc. Aline Marins Paes, UFF

Prof. D.Sc. Eduardo Soares Ogasawara, CEFET/RJ

Niterói

2017

À minha família, meus maiores motivadores.

Agradecimentos

Aos Professores Lúcia Drummond e Daniel de Oliveira, por todo apoio e incentivo recebido desde os tempos da graduação, obrigado pela oportunidade de trabalharmos juntos novamente.

Aos amigos Luan Teylo, Jose Valencia e Rommel Anatoli, que tiveram papel importante no amadurecimento obtido durante o mestrado através da rica troca de conhecimentos.

Aos professores do Instituto de Computação da Universidade Federal Fluminense, por todos os ensinamentos e dedicação.

À CAPES, pela bolsa de mestrado concedida.

Resumo

Workflows científicos são modelos compostos por atividades, dados e dependências entre dados cujo objetivo é representar uma simulação computacional. No geral, estes workflows demandam por muitos recursos computacionais, pois suas execuções envolvem diferentes programas e um grande volume de dados. Desta forma, o uso de ambientes de Computação de Alto Desempenho (CAD) e técnicas de paralelização se tornam essenciais. Nos últimos anos, a Computação em Nuvem vem amadurecendo e se mostrando como uma boa alternativa para a composição destes ambientes, devido a vantagens como facilidade de aquisição, baixo custo inicial e limite de recursos virtualmente inexistente, entre outras. Entretanto, a ocorrência de falhas em tais ambientes deve ser tratada como regra e não como exceção. Existem diversas técnicas que podem ser aplicadas ou adaptadas ao contexto de workflows científicos com o objetivo de tornar suas execuções capazes de suportar falhas com o mínimo impacto possível, tais como reexecução, *checkpoint-restart* e replicação. Este trabalho explora estas técnicas e as analisa aplicadas em casos de workflows científicos reais. Foram implementadas as técnicas *checkpoint-restart* e replicação no Sistema de Gerência de Workflos Científicos (SGWfC) SciCumulus, e estas foram comparadas à técnica de reexecução, técnica mais comumente adotada pelos SGWfC atuais. Por fim, foi realizado um estudo inicial sobre técnicas de Aprendizado de Máquina que auxiliam o cientista na tomada de decisão a respeito de qual técnica seria melhor aplicada a uma determinada tarefa, possibilitando escolhas mais adequadas ao ambiente computacional disponível e a tarefa a ser executada. O uso deste framework de tolerância a falhas melhorou a resiliência do SGWfC SciCumulus ao custo de um reduzido *overhead*, proporcionando a execução de workflows científicos de forma mais estável, rápida e barata - com ganhos de até 30% nos experimentos realizados - mesmo em situações de adversidade.

Palavras-chave: computação em nuvem, computação de alto desempenho, workflows científicos, tolerância a falhas

Abstract

Scientific workflows are models composed of activities, data and dependencies whose objective is to represent a computer simulation. In general, these workflows demand for many computational resources as their executions may involve numerous programs processing a considerable volume of data. Thus, the use of High Performance Computing (HPC) environments allied to parallelization techniques becomes imperative. Over the last years, Cloud Computing has been continuously improved and hence appears as a good candidate to the composition of HPC environments, due to advantages such as easy acquisition, low up-front investiment and virtually non-existent resource limit. However, failure occurrence in such environments is rather a reality than a possibility. There are many fault tolerance techniques that may be applied or adapted to the scientific workflow context, such as re-execution, checkpoint-restart and replication. This work explores these techniques and analyzes their application to real scientific workflow use cases. The checkpoint-restart and replication techniques were implemented in the SciCumulus Scientific Workflow Management System (SWfMS) and compared to the re-execution technique, used as baseline as it is the most commonly adopted fault tolerance technique by current SWfMS. Finally, an initial study on machine learning techniques was conducted in order to assist scientists in the task of choosing the most appropriate fault tolerance technique for a given task, enabling choices well suited for both the task and the currently available computational environment. The use of this fault tolerance framework improved the resilience of the SciCumulus SWfMS at the cost of a low overhead, producing more stable workflow executions in a faster and cheaper manner - with up to 30% improvements obtained at the experiments - even in adverse situations.

Keywords: cloud computing, high performance computing, scientific workflows, fault tolerance techniques

Lista de Figuras

2.1 Workflow CyberShake 5

2.2 Processo de execução de um Workflow Científico 6

4.1 Etapas básicas no tratamento de falhas 21

4.2 Modelo da base de proveniência estendido 23

4.3 Modelo de execução de um workflow no SciCumulus 26

4.4 Arquitetura do *Checkpointer* Distribuído no SciCumulus 28

4.5 Modelo de execução baseado em algebra relacional 31

4.6 Replicação baseada em álgebra relacional 31

5.1 Workflow Montage 35

5.2 Desempenho das TTF para o workflow Montage com 9 atividades e 4 cpus 36

5.3 Desempenho das TTF para o workflow Montage com 9 atividades e 16 cpus 36

5.4 Desempenho das TTF para o workflow Montage com 9 atividades e 22 cpus 37

5.5 Workflow SciPhy 38

5.6 Desempenho das TTF para o workflow SciPhy com 1 genoma de entrada . 39

5.7 Desempenho das TTF para o workflow SciPhy com 4 genomas de entrada . 39

5.8 Desempenho das TTF para o workflow SciPhy com 8 genomas de entrada . 40

Lista de Tabelas

2.1 Principais diferenças entre *Clouds* e *Grids* 8

3.1 Classes de Falhas, de acordo com Cristian [19] 11

3.2 Técnicas de Tolerância a Falhas 13

3.3 Fontes selecionadas 18

3.4 Principais TTF disponíveis nos SGWfC atuais 19

5.1 Tempos de execução das atividades do workflow Montage para diferentes TTF 37

5.2 Ambientes computacionais utilizados para execução do SciPhy 38

5.3 Precisão dos modelos preditivos propostos 43

Lista de Abreviaturas e Siglas

TTF	:	Técnica de Tolerância a Falhas;
VM	:	*Virtual Machine*;
C/R	:	Checkpoint/Restart;
SGWfC	:	Sistemas de Gerência de *Workflows* Científicos;
IaaS	:	Infrastructure-as-a-Service;
PaaS	:	Plataform-as-a-Service;
SaaS	:	Software-as-a-Service;

Sumário

1 Introdução **1**

2 Conceitos Relacionados **4**

2.1 Workflows Científicos 4

2.2 Nuvens Computacionais 7

3 Noções de Tolerância a Falhas **10**

3.1 O que é falha? 10

3.2 Dependabilidade 12

3.2.1 Prevenção de falhas 13

3.2.2 Tolerância a falhas 13

3.2.3 Remoção de falhas 14

3.2.4 Previsão de falhas 14

3.3 Recuperação de Falhas em Workflows Científicos 14

3.3.1 A nível de tarefa 14

3.3.2 A nível de *workflow* 15

3.4 Trabalhos Relacionados 15

3.4.1 Revisão da Literatura 18

4 Tolerância a Falhas no SciCumulus **20**

4.1 Detecção de Falhas 21

4.2 Tolerância a Falhas em Workflows Científicos 26

4.2.1 Retentativa 27

4.2.2 Checkpoint-Restart 27

4.2.3 Replicação de Tarefas 30

5 Resultados Experimentais 34

5.1 Workflow Montage 34

5.2 SciPhy Workflow 38

5.3 Modelo Preditivo para TTF em Workflows Científicos 41

6 Conclusão e Trabalhos Futuros 45

Referências 47

Capítulo 1

Introdução

Workflows científicos podem ser definidos como abstrações utilizadas para se definir sequências de atividades e dependências de dados entre elas [64]. Cada atividade pertencente a um workflow científico representa a invocação de um programa ou um serviço no contexto de um experimento científico [49]. Desta forma, workflows científicos são usados como uma ação controlada em um experimento de forma a confirmar ou refutar uma hipótese científica. Workflows são modelados, executados e monitorados por mecanismos chamados Sistemas de Gerência de Workflows Científicos (SGWfC). O crescimento do uso de workflows científicos e SGWfC nas últimas décadas pode ser percebido em diversos domínios, como biologia, química e astronomia [37]. Muitos destes workflows são ditos de larga escala, ou seja, produzem e consomem uma grande quantidade de dados e, normalmente, são executados repetidamente durante uma quantidade de tempo, até que se confirme ou refute uma dada hipótese científica.

Devido ao grande volume de dados envolvido na execução destes workflows, bem como à alta demanda por processamento, muitos destes workflows precisam ser executados em ambientes de Computação de Alto Desempenho (do inglês High Performance Computing ou, simplesmente, HPC), como *clusters* e *grids* [37]. Desta forma, os SGWfC devem ser capazes de suportar a execução dos workflows em tais ambientes, possibilitando a exploração de oportunidades de processamento paralelo. Muitos dos SGWfC existentes proveem tal capacidade como, por exemplo, Pegasus [25], e-Science Central [36], Chiron [55], Swift/T [69] e SciCumulus [21].

Entretanto, na última década, um novo ambiente de HPC surgiu: a nuvem [66]. Nuvens computacionais oferecem uma vasta variedade de recursos sob demanda, variando desde máquinas virtuais (*virtual machines*, VMs) a armazenamento. Além disso, o ambiente de nuvem oferece seus recursos de forma elástica, o que pode ser de grande valor

para a execução de experimentos científicos de larga escala, uma vez que o experimento pode sofrer variações em suas necessidades. Desta forma, o cientista é capaz de requisitar mais recursos durante a execução de seu experimento ou descartar tais recursos, pagando somente pelo tempo durante o qual os utilizou [48]. Entretanto, a execução de workflows em ambientes de nuvem traz uma série de desafios. Mesmo quando o workflow é executado em VMs de alto desempenho, muitos destes podem ser executados repetidas vezes e cada uma destas execuções pode durar por horas ou até mesmo dias. Especialmente em nuvens, o *datacenter* que hospeda cada uma das VMs pode ser composto de milhares (ou mesmo milhões) de máquinas. Assim, a ocorrência de falhas pode ser dada como certa. De fato, alguns trabalhos já investigaram e discutiram o crescimento na ocorrência de falhas quando a quantidade de recursos utilizados em tal processamento aumenta [42]. Mesmo quando uma VM não falha de fato, seu desempenho está sujeito a variações devido a procedimentos realizados por parte do provedor, como *live migrations*, por exemplo. Se um ou mais recursos falham, o SGWfC deve ser capaz de proceder com a execução do workflow de tal forma que sejam minimizadas as perdas em trabalhos já realizados.

Para possibilitar a recuperação de falhas, muitos programas utilizam mecanismos de tolerância a falhas a nível de aplicação, *i.e.* a aplicação implementa uma estratégia específica a ser adotada para minimizar o impacto de falhas. Entretanto, esta estratégica não pode ser aplicada em workflows científicos. Em muitos workflows, os programas utilizados são caixa-preta, *i.e.* os cientistas não têm acesso ao código fonte. Desta forma, é importante que o SGWfC seja tolerante a falhas, já que não se pode esperar que cada um dos programas que possivelmente comporão um workflow o sejam. Infelizmente, adicionar funcionalidades de tolerância a falhas em SGWfC baseados em nuvem traz uma perda de desempenho adicional. O SGWfC não mais é responsável somente pela gerência da execução do workflow (*e.g* escalonamento de atividades nas VMs, *staging* dos dados, captação e armazenamento de dados de proveniência, *etc.*), mas, também, pelo monitoramento de todo o ambiente, buscando minimizar custos e maximizar a qualidade do resultado final. É importante frisar que estas funcionalidades não mais são opcionais, uma vez que a ocorrência de falhas nestes ambientes pode ser considerada uma regra e não uma exceção.

Muitas Técnicas de Tolerância a Falha (TTF) foram propostas nas últimas décadas [6, 27, 34, 39, 43, 67] e a maioria delas pode ser acoplada aos SGWfC existentes. De fato, algumas abordagens já propõem mecanismos de tolerância a falhas para workflows científicos baseados em nuvem [17]. Entretanto, a maioria destas abordagens propõe reexecuções, em vez de buscar recuperar algum resultado parcial de uma execução prévia. A simples reexecução pode ser uma estratégia eficiente quando o workflow é composto

de muitas atividades de curta duração, *i.e* se tal tarefa falha, a sua reexecução também será de curta duração. Entretanto, quando o workflow é composto de atividades de longa duração, reexecuções acrescentam uma perda de desempenho considerável. Existem outras técnicas, como *Checkpoint-Restart* (C/R) e Replicação que podem ser aplicadas em diversos cenários. O problema é que a melhor técnica a ser aplicada pode variar de acordo com o workflow em questão. Não é possível definir uma só técnica que seja capaz de lidar da melhor forma em qualquer situação, para quaisquer workflows e tratar todo tipo de falhas.

Esta dissertação busca analisar e explorar uma série de Técnicas de Tolerância a Falhas (TTF) em workflows científicos baseados em nuvens computacionais de forma a avaliar a possibilidade de melhorias no tratamento de falhas através da escolha da melhor técnica a ser utilizada, dadas as peculiaridades de um ambiente computacional e das atividades a serem executadas. Foram implementadas e comparadas 3 diferentes TTF no SGWfC SciCumulus: Reexcução, Checkpoint/Restart e Replicação. Foram avaliadas vantagens, desvantagens e o desempenho de cada uma delas em diferentes workflows, em diferentes configurações de ambientes de nuvem. O objetivo destas avaliações é determinar a aplicabilidade destas técnicas em situações diversas, analisando suas qualidades, desempenhos e pontos negativos, além de indicar qual técnica seria mais apropriada para classes específicas de atividades. Assim, a principal contribuição deste trabalho é uma avaliação experimental destas três TTFs em variadas condições de forma que seja possível observar variações de desempenho e, assim, fornecer apoio à decisão dos cientistas sobre a melhor técnica a ser utilizada em execuções reais de seus experimentos.

O restante deste trabalho está organizado da seguinte forma: O Capítulo 2 introduz alguns conceitos que serão importantes ao longo do trabalho, o Capítulo 3 discute noções relacionadas especificamente a técnicas de tolerância a falhas encontrados na literatura, além de trabalhos propostos dentro desta temática. O Capítulo 4 discute a abordagem proposta por este trabalho, que visa tornar um SGWfC capaz de operar mesmo na presença de falhas. O Capítulo 5 apresenta diversos experimentos e analisa os resultados obtidos. Por fim, o Capítulo 6 traz conclusões e discute possibilidades futuras.

Capítulo 2

Conceitos Relacionados

Este capítulo tem por objetivo elucidar características a respeito de importantes conceitos que serão utilizados ao longo deste trabalho, relacionados principalmente a workflows científicos e nuvens computacionais. Serão analisadas suas principais características, usos e desafios associados.

2.1 Workflows Científicos

Com a evolução dos meios computacionais observada nos últimos anos, a computação veio a se tornar tão importante quanto a teoria e a experimentação no método científico [24, 57, 58]. Ambientes altamente distribuídos, tais quais nuvens, *grids* e supercomputadores, passam a tornar possível a simulação de experimentos que antes não seriam praticáveis devido ao tempo despendido ou à quantidade de recursos humanos que haveria de ser empregada e gerenciada. Desta forma, o conceito de workflow científico surge para suprir esta demanda e possibilitar a automatização de simulações científicas, deixando o cientista livre para desenvolver outros aspectos de sua pesquisa [49]. Como exemplos de workflows científicos, pode-se citar SciPhy [54], Montage [8, 13] e CyberShake [12, 33] que tem como objetivo, respectivamente, (i) realizar tarefas relacionadas a análises filogenéticas para desenvolvimento de fármacos, (ii) a criação de mosaicos astronômicos a partir de fotos do espaço tiradas por telescópios, e (iii) calcular a probabilidade de ocorrência de um terremoto em determinada área, todas computacionalmente intensivas e envolvendo uma grande quantidade de dados.

Dá-se o nome de *orquestração* de um workflow científico à atividade de se definir uma sequência de tarefas que deve ser realizada em uma simulação de um experimento científico. O *workflow* em si seria, portanto, o *template* que define a forma com a qual

será feita esta orquestração, e uma *instância* deste workflow é definida por um workflow para um problema em específico juntamente com seus dados de entrada [24].

Um workflow científico possui 4 etapas principais: composição, mapeamento, execução e proveniência [24]. Durante a fase de composição, o cientista define todas as etapas necessárias à execução do seu experimento, além das dependências existentes entre elas, gerando assim um *workflow abstrato*. Estas definições podem ser realizadas de diversas formas, sejam elas textuais (através de XML por exemplo) ou gráficas (auxiliadas por ferramentas específicas para tal fim). Comumente, o resultado desta definição é expressado através de um Grafo Acíclico Direcionado (DAG, do inglês Directed Acyclic Graph), onde tarefas são representadas por nós e arestas representam suas dependências de dados, conforme ilustrado na Figura 2.1. Assim, cada tarefa consumirá dados brutos de entrada ou resultantes do processamento de tarefas predecessoras a si, e gerará uma determinada saída, que poderá ser usada por tarefas sucessoras ou analisada pelo cientista.

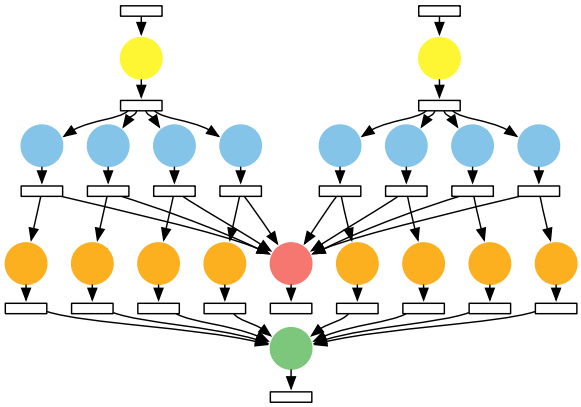

Figura 2.1: Workflow CyberShake

Na fase de mapeamento é definida uma relação tarefa-recurso computacional. Através deste mapeamento é possível a concreta execução de um workflow em recursos computacionais, partindo de uma definição abstrata de workflow. Este mapeamento pode ser realizado manualmente, pelo próprio usuário do workflow, ou automaticamente, por um sistema responsável pela gerência do experimento. Neste caso, o mapeamento pode ser realizado de forma a otimizar o tempo de execução do workflow ou, ainda, custos associados com a utilização de recursos em nuvem, por exemplo [18, 24, 65].

Na chamada fase de execução são definidos modelos de execução e ocorre o escalonamento e execução das tarefas conforme o mapeamento realizado na fase anterior, ainda que seja possível a realização de adaptações em tempo de execução, de forma a otimizar aspectos como tempo de execução e evitar, ou, ao menos, aliviar a ocorrência de falhas.

Em termos de possíveis modelos de execução, pode-se por exemplo optar por simples execuções sequenciais ou, ainda, execuções paralelas, se possível. Para mitigar a ocorrência de falhas podem ser aplicadas diversas técnicas, visando tanto a prevenção quanto a restauração do correto estado do sistema, conforme será visto em mais detalhes no Capítulo 3.

A fase de proveniência é a fase durante a qual são coletados os metadados do experimento [20, 49]. São coletadas, por exemplo, informações a respeito de qual tarefa foi a responsável pela criação de cada um dos dados de entrada ou saída das diversas tarefas, por quanto tempo uma tarefa foi executada, quais recursos foram utilizados no processamento de uma tarefa, qual tarefa consumiu um determinado dado, etc. Tais informações são primordiais para permitir a reprodutibilidade e consequente validação do experimento. Dadas todas estas atividades que cercam a execução de um workflow científico, surge a necessidade de sistemas que apoiem a gerência do ciclo de vida dos mesmos, desde a fase de composição até a análise dos resultados obtidos. Sistemas de Gerência de Workflows Científicos (SGWfC) são sistemas que visam fornecer ferramentas que auxiliem os cientistas nas diferentes fases deste ciclo. Como exemplos de tais SGWfC temos o Kepler [1], Pegasus [23], SciCumulus [21], Swift/T [68], Taverna [56], Triana [63], entre outros.

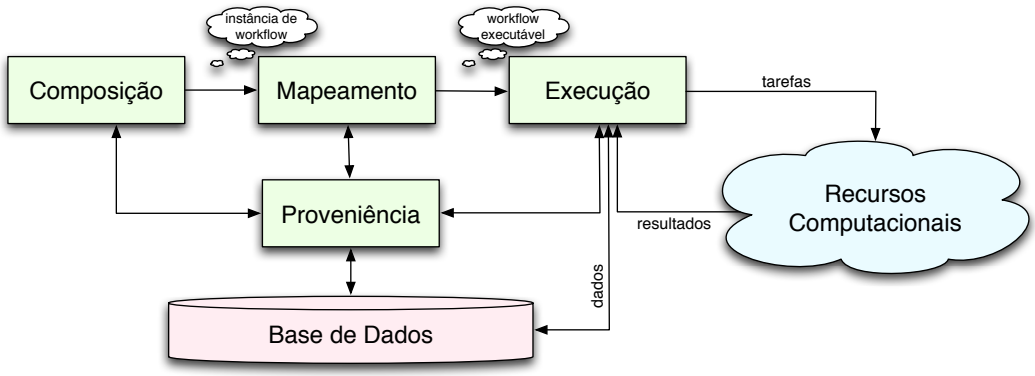

Figura 2.2: Processo de execução de um Workflow Científico

Deelman et al. [22] relata os principais desafios a serem enfrentados em busca de melhorar a usabilidade de workflows científicos. Alguns importantes aspectos seriam, por exemplo, a melhoria da usabilidade, com a oferta de opções que possibilitassem a tomada de decisões durante a execução dos experimentos, através de alterações em parâmetros ou, até mesmo, no curso da execução. Também é fundamental que o experimento possa ser reproduzido e, desta forma, suas conclusões possam ser facilmente confirmadas; a linguagem a ser utilizada deve ser amigável para o cientista, uma vez que este não necessariamente terá um profundo entendimento sobre assuntos além do domínio do seu

estudo, etc. Finalmente, uma das funções principais dos sistemas executores de workflows científicos seria a capacidade em executá-los de forma correta e confiável - principal alvo de estudo deste trabalho. Ainda de acordo com Deelman *et al.* [22], e como será visto ao longo deste trabalho, a detecção e tolerância a falhas na execução de workflows científicos poderia ser realizada de forma mais adequada ao melhor aproveitamento dos recursos computacionais disponíveis do que o observado atualmente nos SGWfC mais populares.

2.2 Nuvens Computacionais

Computação em nuvem tem sua origem a partir de uma antiga ideia: computação utilitária (do inglês *utility computing*) [2, 60]. Da mesma forma que diversos serviços públicos, como o fornecimento de água, energia elétrica, telefonia, transporte e acesso à internet, percebeu-se que seria vantajoso o oferecimento de recursos computacionais sob demanda na forma de serviços.

Estes serviços são caracterizados por uma série de requisitos, sem os quais a demanda dos usuários não seria atendida. Por exemplo, um serviço de transporte público deve ser minimamente *confiável* em termos de frequência e trajetos, uma vez que, sem a certeza de que será levado ao seu destino no momento em que precisa, tal serviço de nada serviria aos seus usuários. Ainda levando em consideração o exemplo do transporte, não seria vantajoso que ônibus e trens fossem completamente substituídos por opções de transporte particular, já que isto inevitavelmente traria um grande aumento no valor deste serviço. Assim, a *escalabilidade* tende a trazer benefícios tanto para o fornecedor quanto para o consumidor deste serviço.

Desta maneira, temos que a Computação em Nuvem, inspirada na ideia de computação utilitária, pode ser definida como o fornecimento de serviços na forma de aplicações ou acesso a *hardware* através da Internet, sob demanda [2]. Este paradigma pode ser comparado à computação em grades, apesar de existirem algumas características fundamentais que as diferenciam, principalmente em termos de compartilhamento de recursos, centralização de controle do provedor, usabilidade e modelo de pagamento [11, 66]. A Tabela 2.1 traz as principais diferenças entre estes modelos de computação.

De forma mais consistente, Vaquero et al. [66] define a computação em nuvem como um grande conjunto de recursos virtualizados, de fácil uso e acessíveis, variando desde *hardware* até aplicações fornecidas como serviços. A provisão destes recursos deve ocorrer de forma escalável e sob demanda, de forma a atender quaisquer variações nas necessidades

	Cloud	***Grid***
Domínio	Centralizado	Descentralizado
Pagamento	Flexível, pagamento por uso	Rígido, taxas fixas, financiamento público
Recursos	Não são compartilhados[1]	Compartilhados pelos membros
Usabilidade	Amigável	Difícil de ser utilizado
Interconexão	Baixa latência	Alta latência
Falhas	Replicação de dados e migração de VMs	Limitada, correção por reinicialização

Tabela 2.1: Principais diferenças entre *Clouds* e *Grids*

[1]Recursos, enquanto atribuídos a um usuário, não são compartilhados com outros usuários

dos seus usuários. E, a exemplo dos serviços públicos mais tradicionais, os custos são calculados com base no seu uso.

Os serviços de nuvem, de modo geral, se enquadram em um dos três modelos a seguir (ordenados do mais alto nível para o mais baixo nível):

- **Software-as-a-Service (SaaS)**: Nesta categoria se enquadram softwares que são fornecidos aos usuários utilizando-se os conceitos de computação em nuvem, ou seja, provisionados através da Internet, sob demanda, de forma escalável. Alguns exemplos de software como serviço são o Dropbox[1] e o Gmail [2], entre outros.

- **Plataform-as-a-Service (PaaS)**: Aqui se encontram serviços que proveem recursos comumente necessários para o desenvolvimento das aplicações finais, como bancos de dados, autenticação, servidores web, etc. Como exemplo pode-se citar o Google App Engine [3] e Heroku[4], ambos voltados ao desenvolvimento de aplicações web e que facilitam a publicação destas aplicações através do fornecimento de servidores web e de DNS, serviço de controle de versão, auto balanceamento de carga, etc.

- **Infrastructure-as-a-Service (IaaS)**: Se refere ao fornecimento de recursos computacionais em um nível mais próximo ao metal puro. São fornecidos processadores, memória RAM, discos de armazenamento e interconexão de rede. A partir destes recursos disponibilizados, o usuário pode combiná-los da forma que melhor lhe convir para realizar a tarefa de seu interesse. Como exemplos desta classe de serviços temos o Amazon Elastic Cloud Computing [5], Amazon Simple Storage Service [6],

[1]https://dropbox.com
[2]https://gmail.com
[3]https://appengine.google.com/
[4]https://www.heroku.com/
[5]https://aws.amazon.com/pt/ec2/
[6]https://aws.amazon.com/pt/s3/

Google Compute Engine[7], Google Cloud Storage[8], etc.

Este trabalho visa estudar as formas de se aproveitar a disponibilidade de tais recursos aplicados à execução de experimentos científicos de larga escala. A aliança das técnicas de composição e gerência de execução de workflows ao poder computacional e à conveniência no acesso à Computação em Nuvem permite que os cientistas executem seus experimentos de forma mais rápida e barata. Entretanto, há de se levar em consideração que, devido ao emprego de grandes quantidades de recursos em tais experimentos, a probabilidade de ocorrência de uma falha em um recurso qualquer aumenta, tornando necessário o estudo das diferentes ameaças às quais o experimento está sujeito e a melhor forma de abordá-las. O capítulo a seguir trata das diferentes formas de falhas observadas em aplicações computacionais, bem como os principais meios de mitigação e prevenção das mesmas, principalmente no contexto de nuvens computacionais e de workflows científicos.

[7]https://cloud.google.com/compute/?hl=pt-br
[8]https://cloud.google.com/storage/?hl=pt-br

Capítulo 3

Noções de Tolerância a Falhas

O interesse por tolerância a falhas em sistemas computacionais e de telefonia remonta ao início da utilização dos computadores para telefonia, antes mesmo da utilização de máquinas de controle por software armazenado (*SPC - Stored Program Control machines*). Desde o início da utilização de tais sistemas, percebeu-se que sistemas de grande complexidade nunca são livre de falhas [4].

Desta forma, tolerância a falhas é um conceito chave quando se executam aplicações em ambientes altamente distribuídos - como supercomputadores, *Grids* e Nuvens - pois a probabilidade de falhas cresce conforme aumenta o número e a heterogeneidade dos recursos envolvidos em tais sistemas [42].

Este capítulo visa elucidar diversos conceitos importantes que serão utilizados ao longo deste trabalho, além de identificar falhas que ameaçam um sistema, suas principais fontes, como podem afetar o funcionamento do sistema, como preveni-las e recuperar os sistemas dada a detecção de uma falha.

3.1 O que é falha?

Inicialmente, é importante haver entendimento a respeito do conceito de falha. Falha, erro e falta são termos que são utilizados na literatura de forma ambígua, podendo denotar desde os níveis mais baixos de abstração (*e.g.* problemas de memória) até níveis mais altos tais quais comportamentos incorretos de um dado sistema em termos de suas especificações [5, 52, 32]. Entretanto, a *falha* pode ser entendida como o evento que ocorre quando o sistema não se comporta da maneira devida, ou seja, faz uma transição para um estado de *erro*. Já a *causa* da falha, seja algorítmica ou mecânica, pode ser chamada de *falta*

[50].

Pode-se dizer que uma aplicação funciona corretamente se sua resposta se dá de acordo com o especificado e dentro do tempo estabelecido e diz-se que um sistema, ou aplicação, falhou caso contrário. A falha pode classificada de acordo com suas características de diversas formas. Cristian [19] fornece uma possível classificação. A Tabela 3.1 traz estas classificações de falhas e suas respectivas características.

Tabela 3.1: Classes de Falhas, de acordo com Cristian [19]

Classe	**Comportamento**	**Possíveis Causas**
Falha por omissão	A aplicação não emite qualquer resposta a uma dada entrada	- não recebimento de requisição - perda de mensagens
Falha de valor	O valor da resposta dada pela aplicação é incorreta	- erro de implementação - dado de entrada incorreto
Falha de transição	A aplicação realiza uma transição de estado incorreta	- erro de hardware - falta de espaço em disco
Falha de desempenho	A aplicação responde corretamente, porém fora do intervalo de tempo previsto	- sobrecarga de processamento - sobrecarga na rede - *deadlock*
Crash	Após uma falha por omissão, a aplicação não mais emite respostas para quaisquer requisições até que seja reinicializada	- erro de hardware - erro de software

Os sistemas computacionais podem ainda ter suas falhas classificadas como sendo do tipo *crash*, onde os processadores param de funcionar em determinado momento; *fail-stop*, em que um processador falha e esta situação pode ser verificada a partir de seus vizinhos; e *bizantinas*, onde processadores passam a agir de maneira arbitrária.

Nelson [52] classifica falhas de acordo com diversos aspectos como duração e extensão, por exemplo. Em relação à duração, as falhas podem ser classificadas como transitórias, intermitentes ou permanentes. Estados de falha transitórios são aqueles que ocorrem durante um tempo finito e não são recorrentes. Geralmente são causados por fatores externos, como a execução de rotinas de *live-migration*, por exemplo. Já os estados de falha intermitentes são aqueles que ocorrem durante tempo finito mas são recorrentes, *i.e.* o sistema oscila entre seu estado correto e estados incorretos. Uma possível causa de estados de falha intermitentes é a variação na demanda por recursos do sistema. Por fim, estados de falha permanentes são aqueles onde não pode ser observado um estado de operação correto em qualquer tempo após a ocorrência da falha. Podem ser decorrentes de erros de implementação, falhas de hardware, etc. Já em termos de extensão, falhas podem ser classificadas como *locais* ou *globais*, denotando, respectivamente, falhas que

afetam componentes individuais ou múltiplos componentes.

Lamport [45] define duas classes de propriedades principais de sistemas multiprocessos necessárias à garantia de sua corretude: *safety* e *liveness*. Propriedades do tipo *safety* definem que algo não deverá ocorrer, enquanto propriedades do tipo *liveness* definem que algo deverá ocorrer em algum momento. Em uma execução de workflow científico tolerante a falhas, representado por um Grafo Acíclico Direcionado (DAG, do inglês Direct Acyclic Graph), *safety* define que nenhuma tarefa pode terminar sua execução sem gerar sua respectiva saída, enquanto *liveness* define que todas as tarefas deverão encontrar um determinado dado de entrada e ser executadas. Desta forma, tais propriedades garantiriam a correta terminação da execução de um workflow científico.

3.2 Dependabilidade

Dependabilidade (do inglês *dependability*) é um conceito que define a habilidade de um dado sistema em oferecer um serviço justificadamente confiável, evitando falhas mais frequentes e graves do que o aceitável [5, 46]. Este conceito engloba outros atributos como disponibilidade (o sistema deve estar pronto para ser usado quando necessário), confiabilidade (baixa ocorrência de interrupções no fornecimento do serviço), segurança (não ocorrência de catástrofes a partir de falhas no sistema), integridade (dados não sofrem alterações impróprias) e manutenibilidade (facilidade em se realizar reparos), e, de acordo com o objetivo deste sistema, a cada um destes pode ser dada maior ou menor ênfase. A dependabilidade pode ser alcançada através de técnicas que podem ser agrupadas em 4 principais categorias: prevenção de falhas, tolerância a falhas, remoção de falhas e previsão de falhas.

De forma geral, sistemas computacionais possuem seu ciclo de vida composto por duas fases principais: *desenvolvimento* e *uso*. Em ambas as fases, o sistema está exposto a diversas vulnerabilidades. Por exemplo, há, durante a fase de desenvolvimento, o risco de o sistema não ser desenvolvido de acordo com suas especificações devido a imperícia por parte dos desenvolvedores ou, ainda, por problemas nas próprias especificações. Já durante a fase de uso do sistema, existem os riscos inerentes aos ambientes nos quais este sistema será executado, podendo vir a sofrer danos decorrentes de mau uso, falhas em outros sistemas do qual este venha a depender, falhas na infraestrutura que o suporta, entre outros.

A seguir serão vistas em mais detalhes cada uma das grandes categorias de técnicas

envolvidas na dependabilidade.

3.2.1 Prevenção de falhas

A prevenção de falhas é realizada através de técnicas que têm por objetivo principal tornar o sistema menos inclinado à ocorrência de falhas. A maior parte do trabalho realizado neste sentido se dá na fase de desenvolvimento de um sistema, antes do mesmo entrar em uso, através de técnicas que se baseiam fundamentalmente na melhoria dos processos de desenvolvimento, com técnicas de validação dos requisitos através de modelos, análises de complexidade, modelos de transição de estados, etc [9, 15, 51].

3.2.2 Tolerância a falhas

Um sistema pode ser dito tolerante a falhas se permanece em um estado correto mesmo após a ocorrência de erros [3]. Para alcançar tal comportamento, é necessário que o sistema seja projetado tendo em vista as falhas que possam vir a ocorrer e, assim, tenha incorporado mecanismos para lidar com estados incorretos, evitando que venham a se manifestar em falhas [50].

Tolerância a falhas é alcançada através dois procedimentos principais, que são complementares: detecção de erros e recuperação do sistema. A Tabela 3.2 resume as principais técnicas relacionadas a cada um destes procedimentos.

Tabela 3.2: Técnicas de Tolerância a Falhas

Detecção de Erros		**Recuperação do Sistema**			
		Técnicas Reativas		Técnicas Preventivas	
Detecção Simultânea	Busca identificar falhas durante a execução normal do sistema	*Rollback*	Traz o sistema a um estado (*checkpoint*) consistente gravado antes da ocorrência do erro	Diagnose	Consiste em identificar, classificar e localizar a causa do erro de forma que este possa ser evitado no futuro
		Rollforward	Semelhante ao *rollback*	Isolamento	Realiza o isolamento físico ou lógico do componente defeituoso
Detecção Preemptiva	Busca identificar falhas enquanto o sistema está em estado suspenso	Compensação	Solução de falhas através de redundância	Reconfiguração	Faz uso de componentes reserva ou redistribui tarefas entre componentes livres de falha
				Reinicialização	Armazena configurações de estados consistentes e atualiza o sistema para tais

Diversos trabalhos já foram realizados tanto no sentido de se detectar a ocorrência das falhas quanto para corrigi-las. O algoritmo de *heartbeat* [44], amplamente conhecido, é um exemplo de técnica de detecção simultânea de falhas em componentes. Já em relação

às técnicas de recuperação, pode-se citar o algoritmo de Chandy-Lamport [14] (*rollback recovery*), técnicas de *software rejuvenation* [40] (reinicialização), o protocolo *primary-backup* [10], replicação de estado de máquina [62] (compensação por redundância), entre outras.

3.2.3 Remoção de falhas

A remoção de falhas pode ser realizada durante a fase de desenvolvimento dos sistemas, fazendo-se uso de diversas técnicas de testagem, ou mesmo após o início do uso dos mesmos, em manutenções preventivas ou corretivas.

3.2.4 Previsão de falhas

A previsão da ocorrência de uma falha em um dado sistema em execução pode ser realizada através da análise e avaliação do comportamento e de métricas deste sistema ao longo do tempo, possibilitando desta forma a determinação de componentes que estariam em potencial risco de ocorrência de falhas.

3.3 Recuperação de Falhas em Workflows Científicos

Diversas técnicas são aplicadas na recuperação de falhas em workflows científicos. Alguns trabalhos [32, 41, 71] se propuseram a classificá-las de acordo com seu método de recuperação e o nível no qual a falha é tratada: a nível de *workflow* ou a nível de tarefa. Técnicas a nível de tarefa são aquelas em que a recuperação é realizada a nível de aplicação para encobrir os efeitos de uma falha. Já as técnicas de recuperação de falhas a nível de *workflow* são aquelas na qual são realizadas alterações na estrutura do *workflow*, de forma que o mesmo esteja melhor preparado pra lidar com a ocorrência de falhas. A seguir serão vistas as definições das principais técnicas de tolerância a falhas, agrupadas de acordo com suas classes.

3.3.1 A nível de tarefa

- **Re-tentativa**: Nesta técnica, também referida por re-execução ao longo deste trabalho, é realizada uma tentativa de execução da mesma tarefa no mesmo recurso após a ocorrência de uma falha (mesmo processador ou mesma VM).

- **Recurso alternativo**: Esta técnica é semelhante à técnica de re-tentativa, porém a nova tentativa de execução da tarefa é realizada em outro recurso.
- **Checkpoint-Restart (C/R)**: Dados de estados parciais da execução são armazenados periódicamente de forma que, após a ocorrência de uma falha, seja possível retomar o processamento em uma máquina qualquer restaurando-se o último estado consistente (conjunto de valores em memória) gravado.
- **Replicação**: Esta técnica se vale da execução concomitante de uma mesma tarefa em N recursos diferentes, de forma a tolerar N-1 falhas.

3.3.2 A nível de *workflow*

- **Tarefa alternativa**: Esta técnica consiste em definir outra tarefa ou, até mesmo, um ramo independente no DAG que pode ser executado no lugar de uma tarefa que falhou, utilizando as mesmas entradas e gerando saídas compatíveis.
- **Redundância**: Semelhante às técnicas "tarefa alternativa" e "replicação"; entretanto, neste caso, diferentes tarefas são executadas ao mesmo tempo de forma que a execução com sucesso de ao menos uma das tarefas redundantes garanta a correta execução do passo em questão no experimento.
- ***Rescue workflow***: São armazenados resultados já gerados com sucesso de forma que uma execução do workflow possa ser reiniciada após uma falha qualquer sem que haja necessidade de se re-executar tarefas já finalizadas corretamente.
- **Exceção definida por usuário**: Esta técnica permite que o usuário do SGWfC especifique rotinas para lidar com um dado tipo de falha.

3.4 Trabalhos Relacionados

Conforme dito anteriormente, tolerância a falhas é um conceito chave na execução de workflows científicos em ambientes altamente distribuídos. Assim, diversos trabalhos já investigaram e propuseram soluções para este tópico. Estes trabalhos buscam tanto prevenir a ocorrência de falhas quanto recuperá-las, dada sua ocorrência.

Em termos de prevenção e detecção de falhas, Gupta [35] desenvolveu um *framework* no qual agentes ficam responsáveis por receber informações a respeito de falhas nos diversos componentes do sistema (enviadas pelos próprios componentes no qual a falha veio a

ocorrer) e difundir tal informação entre os demais componentes, de forma que cada um possa tomar as devidas providências individualmente.

Hu et al. [39] propôs uma heurística de escalonamento que é capaz decidir replicar tarefas em certas situações, como ao detectar recursos ociosos, por exemplo. O escalonamento de réplicas, especialmente de tarefas pertencentes ao caminho crítico do workflow (ramo do DAG que limita o tempo mínimo de execução do workflow), é interessante pois a ocorrência de uma falha em alguma destas tarefas ou, ainda, simples lentidão no processamento das mesmas, tende a impactar todo o restante do workflow. Com isso, a inclusão destas tarefas redundantes permite que falhas ou lentidão sejam suportadas com mínimo impacto.

Gu et al. [34] se propôs a resolver o problema de mapeamento de recursos para um workflow científico enquanto minimiza a ocorrência de falhas e maximiza a vazão (*throughput*). Apesar de mencionar técnicas de tolerância a falhas conhecidas na literatura, tais como re-tentativa, replicação de tarefas e *checkpoint-restart*, a solução apresentada se baseia no escalonamento de tarefas em recursos cuja taxa de falhas é baixa, minimizando, por consequência, a taxa de falhas nas tarefas.

Bala e Chana [6] apresentam, também, uma abordagem baseada em escalonamento. No entanto, esta abordagem leva em consideração a probabilidade de se observar uma falha devido a superutilização dos recursos de determinado *host*. Desta forma, é possível melhorar a utilização de recursos priorizando-se o escalonamento de tarefas com o maior número de dependências e, consequentemente, reduzindo o tempo médio de execução do workflow. Para evitar falhas, VMs podem ser migradas para *hosts* com menor taxa de utilização. Entretanto, esta abordagem é impraticável do ponto de vista dos usuários de nuvens públicas, uma vez que provedores destes serviços como Amazon Web Services (AWS), Google Compute Engine (GCE) ou Microsoft Azure geralmente não permitem este tipo de controle aos seus clientes.

Jain et al. [43] apresentam seu *framework* chamado FireWorks que visa prover uma plataforma tolerante a falhas para a execução de workflows científicos. Esta plataforma contém diversas funcionalidades, permitindo a coleta de dados de proveniência, execução de tarefas concorrentes, intervenções em workflows em execução, etc. Em termos de tolerância a falhas, os autores detectam 3 tipos de falhas: "*soft*" (o script retorna um erro), "*hard*" (a máquina para de funcionar) e falhas de fila. Para resolver falhas do tipo "*soft*", os usuários precisam examinar os problemas da ativação, corrigí-los manualmente e, por fim, ressubmetê-los ao sistema. Falhas "*hard*" são detectadas por um sistema de *heartbeat*.

Novamente, a única técnica de tolerância a falhas mencionada e/ou implementada é re-tentativa. O último tipo de falha considerado é o de fila, onde a ativação pode se perder no sistema de filas ao qual foi submetida, após sua remoção por parte dos administradores, por exemplo. Para resolver este problema, o SGWfC provê uma rotina através da qual os usuários podem ressubmeter ativações que estão na fila por um período de tempo mais longo que um parâmetro especificado.

Desta forma, percebe-se que a maior parte dos SGWfC normalmente se baseia em algumas das técnicas vistas na Subseção 3.3. Re-tentativa e Replicação aparecem como as técnicas mais populares, como em [7, 29, 30, 63]. Apesar de as técnicas baseadas em C/R também serem comuns, normalmente são aplicadas a nível de workflow [38, 72] (caracterizando assim algo semelhante a um *rescue workflow*). Entretanto, esta técnica não é capaz de recuperar o trabalho parcial realizado por uma tarefa que falhou. Em se tratar de uma tarefa de longa duração, esta perda viria a estender o *makespan* global da execução, já que todas as tarefas dela dependentes teriam seu tempo de espera aumentado, e aumentar os custos referentes ao serviço de nuvem.

Em relação às técnicas de C/R a nível de aplicação, os SGWfC geralmente se baseiam na inclusão de bibliotecas específicas para tal fim no código fonte das aplicações [27, 67]. Esta solução, apesar de realmente fornecer a capacidade de restauração de *checkpoints*, se mostra inviável no contexto de workflows científicos uma vez que estes, normalmente, são compostos de diversos programas caixa-preta providos por terceiros.

Alguns trabalhos, como Li *et al.* [47] e Engelmann *et al.* [28], mostram a relevância da utilização de técnicas de previsão como forma de auxílio à execução de aplicações na presença de falhas. Li *et at.* [47], por exemplo, desenvolveu um mecanismo de tolerância a falhas que não se compromente a somente corrigir uma falha, mas busca também, através de um modelo de decisão baseado em custos, decidir pela adoção da medida de prevenção de falhas mais adequada a determinado momento, visando a redução do tempo de execução das aplicações. Entretanto, o trabalho foca mais no modelo de custos que no modelo de previsão, não havendo muitos detalhes a respeito de como são obtidos os valores de previsão de ocorrência de falhas ou que dados suportam tal modelo preditivo. O trabalho de Engelmann *et al.* segue um caminho parecido, ao desenvolver as fundações necessárias para a tolerância a falhas em ambientes de CAD fazendo uso de previsões de falhas através de históricos de execução e aprendizado de máquina. Entretanto, assim como Li *et al.*, elabora um trabalho de fundamentação teórica, sem apontar estratégias concretas que pudessem ser avaliadas e aplicadas em casos semelhates.

Já Fu *et al.* [31], por outro lado, implementou um *framework* de previsão de falhas que utiliza informações de traces de execuções em *clusters* e supercomputadores, além de informações de alocação de aplicações, para calcular correlações espaciais e temporais entre falhas. A partir destes cálculos foi possível prever com alta acurácia a localização e o momento de ocorrência de uma falha e, com isso, tomar as devidas medidas preventivas.

3.4.1 Revisão da Literatura

Para melhor ilustrar as atuais capacidades dos sistemas de gerência de workflows científicos atuais, uma revisão da literatura foi realizada. Buscou-se responder duas questões principais:

- Quais SGWfC consideram questões relacionadas a tolerância a falhas?
- Quais técnicas de tolerância a falhas eles utilizam?

Três bases de dados eletrônicas (Tabela 3.3) foram selecionadas baseadas nos seguintes critérios:

- Ter publicação de artigos regularmente atualizada
- Disponibilizar todos os artigos para análise
- Submeter todos os artigos a processo de revisão por pares

Tabela 3.3: Fontes selecionadas

Base	**URL**
ACM Digital Library	http://dl.acm.org
Google Scholar	https://scholar.google.com
IEEE Xplore	http://ieeexplore.ieee.org

As bases selecionads foram consultadas utilizando-se a seguinte *string* de consulta: **"(resilience OR resilient OR (fault AND (tolerance OR tolerant)) AND (scientific AND workflow AND management AND system)"**. Do total de resultados, foram analisados os 40 mais relevantes retornados pelo Google Scholar e todos os demais obtidos nas bases ACM e IEEE, em um total de 65 resultados. Estes foram analisados e os estudos que não abordavam qualquer implementação de tolerância a falhas ou não envolviam um SGWfC foram descartados. Assim, a Tabela 3.4 resume os resultados obtidos, mostrando alguns dos SGWfC mais populares durante o desenvolvimento

desta dissertação, juntamente com suas respectivas abordagens em termos de tolerância a falhas.

É possível perceber a partir desta tabela que:

- 53,8% dos SGWfC implementam a técnica Re-tentativa
- 46,1% dos SGWfC implementam a técnica *Checkpoint-Restart*
- 65,3% dos SGWfC disponibilizam somente uma Técnica de Tolerência a Falhas (TTF)

Entre os SGWfC que proveem a funcionalidade de *Checkpoint-Restart*, a maior parte se baseia em implementações a nível de aplicação, enquanto o restante faz uso de *checkpoint* a nível de workflow (equivalente à técnica *rescue workflow*).

Tabela 3.4: Principais TTF disponíveis nos SGWfC atuais

SGWfC	**Re-tentativa**	**Checkpoint**	**Replicação**	**Tarefa Alternativa**	**Intervenção de usuário**
ActiveBPEL					✓
Askalon	✓	✓			
Chemomentum	✓	✓			
Chiron	✓				
DVega				✓	
EPSWFlow	✓				
Escogitare					
FT-SWF	✓				
GridWay		✓			
GridFlow		✓			
GWEE		✓			
GWES		✓			
Kepler	✓	✓		✓	
Kepler + Hadoop			✓		
Heracles					✓
P-GRADE	✓	✓			
Pegasus	✓	✓			
ProActive	✓	✓			✓
SciCumulus	✓				
Swift/T	✓	✓			
Taverna	✓				
Triana	✓			✓	
Trident		✓			
Unicore5	✓				
Vistrails					
VGrADS			✓		
YAWL					✓

Capítulo 4

Tolerância a Falhas no SciCumulus

Neste capítulo serão apresentadas as diferentes técnicas que foram aplicadas ao SGWfC SciCumulus com o objetivo de se detectar e recuperar diferentes tipos de falhas na execução dos workflows, aumentando assim a resiliência deste SGWfC. Conforme visto na Seção 3.3, as técnicas de recuperação podem ser divididas de acordo com o nível no qual operam: a nível de workflow ou a nível de tarefa. Algumas das técnicas a nível de workflow requerem que a estratégia de tolerância a falhas seja definida pelo usuário durante a fase de composição do workflow, com a especificação de tarefas alternativas ou redundantes, por exemplo. Estas não farão parte do escopo deste trabalho pois um dos objetivos aqui buscados é fornecer aos cientistas um sistema capaz de processar um workflow científico utilizando recursos de nuvens computacionais e tratando possíveis falhas de forma transparente para o mesmo, que, desta forma, poderá elaborar seu experimento com maior facilidade.

A técnica *rescue workflow*, ainda que classificada como a nível de workflow, não requer intervenção do cientista. Pode ser implementada através da simples gravação dos passos executados pelo workflow (tarefas finalizadas com sucesso) e armazenamento dos produtos intermediários gerados (dados de saída das tarefas). Tal estratégia é de grande valia na recuperação de falhas em SGWfC pois evita que boa parte do trabalho já realizado não precise ser refeito em caso de falhas que impeçam o prosseguimento da execução do workflow. Esta estratégia, no entanto, não é capaz de recuperar falhas ocorridas em tarefas de longa duração (muito comuns em workflows científicos) sem que haja a perda do esforço já realizado até o momento anterior à ocorrência da falha. Com isso, esta estratégia também não é alvo de estudos mais aprofundados neste trabalho.

Este trabalho, portanto, busca aumentar a dependabilidade de um SGWfC através de um framework de tolerância a falhas, sejam elas transitórias, intermitentes até mesmo

permanentes, causadas pela oscilação dos recursos de hardware fornecidos pelas nuvens computacionais. A Figura 4.1 ilustra, em alto nível, as etapas através das quais se busca tornar um SGWfC capaz de lidar com falhas. Conforme pode ser visto, são necessários esforços visando (i) detectar a falha (proativa ou reativamente), (ii) coletar dados de proveniência para suportar tanto a detecção quanto a (iii) recuperação, que pode ser realizada através de diferentes técnicas. As subseções a seguir trazem mais detalhes a respeito de como cada uma destas etapas foi implementada no SGWfC SciCumulus.

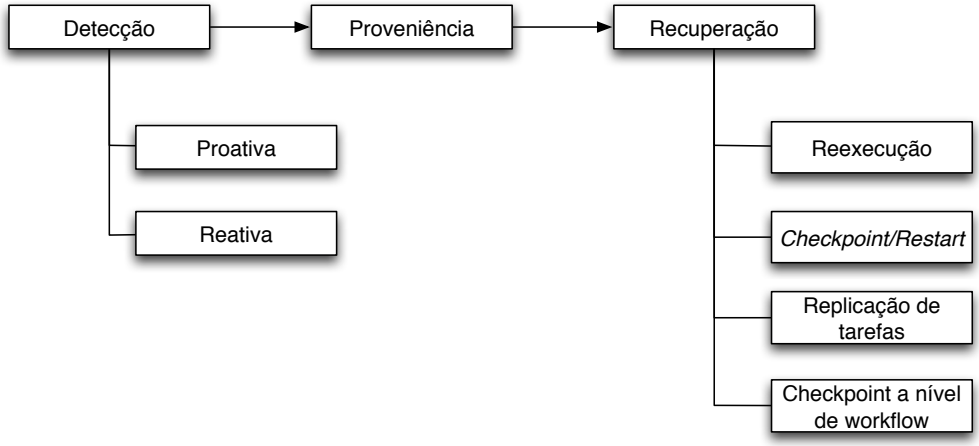

Figura 4.1: Etapas básicas no tratamento de falhas

4.1 Detecção de Falhas

A detecção de falhas deve ser o primeiro passo a ser tomado ao se buscar aumentar a dependabilidade de um sistema. Alguns trabalhos como Nguyen *et al.* [53] e Gupta *et al.* [35] abordam este tema e propõem soluções neste sentido, conforme visto na Seção 3.4. De fato, a tal difusão da informação proposta por Gupta *et al.* é muito importante para que as aplicações possam agir e evitar uma possível falha ou a corrigir. Entretanto, em um contexto de workflows científicos, onde grande parte das aplicações envolvidas foi elaborada por terceiros e são caixa-preta, não se pode esperar que haja integração com um sistema de difusão de informações sem que seja feita alguma implementação neste sentido, o que muitas vezes pode se tornar inviável para o pesquisador.

Este trabalho busca realizar a detecção de falhas a partir de duas frentes principais. Na primeira, de forma semelhante ao que foi proposto por Gupta, o SGWfC é capaz de receber informações referente às aplicações a respeito da ocorrência de falhas. Estas informações, entretanto, podem ser simplesmente o parâmetro de saída padrão do programa,

não havendo necessidade de qualquer tipo de integração mais profunda entre a aplicação e o SGWfC, que envolva alterações na aplicação, por exemplo. O SGWfC armazena uma série de informações sobre as execuções, tais quais:

- Quais aplicações já foram executadas
- Data e hora de início e término de cada uma das execuções
- Qual máquina foi utilizada para cada uma das execuções
- Número de tentativas de execução realizados

Estas informações são armazenadas na base de dados de proveniência do SGWfC, de forma que seja possível rastrear diversos aspectos relacionados a uma determinada execução de uma tarefa do workflow. A base de dados do SciCumulus foi estendida para suportar o armazenamento de todo o histórico de falhas ocorridas no sistema e, assim, ser possibilitar consultas como:

- qual tarefa vem sofrendo uma quantidade de erros acima de determinado limiar?
- qual máquina está sofrendo um alto número de falhas, independentemente da tarefa sendo executada?

Estas informações podem ser um indicativo de que algo não está correto e é necessário algum tipo de intervenção.

A segunda frente se dá na forma de monitoramento do ambiente, através de coleta de informações a respeito da "saúde" das máquinas. Estas informações, por vezes, são disponibilizadas pelos provedores de computação em nuvem através de APIs, como no caso da Amazon[1] ou podem ser obtidas a partir da instalação de softwares de monitoramento. Este trabalho utilizou as informações fornecidas pela Amazon. Através de chamadas a API disponibilizada pela mesma, é possível consultar periodicamente informações a respeito do ambiente computacional em uso. São fornecidas diversas informações como, por exemplo, as taxas de uso de processador, disco, rede e memória, além do estado (correto funcionameno ou com algum tipo de problema de conectividade, desempenho, etc) de cada um destes recursos.

A Figura 4.2 mostra o modelo E-R da base de dados que suporta o SciCumulus e as alterações que foram realizadas para armazenar informações referentes à detecção de falhas

[1]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring_ec2.html

em tarefas e no ambiente. Foram incluídas as seguintes tabelas: *eactivation_error_log* e *instance_status_check* (a tabela *checkpoint_log* será comentada mais a frente), que armazenam, respectivamente, informações de erros ocorridos nas execuções das tarefas do workflow e de erros ocorridos nas instâncias das máquinas em nuvem.

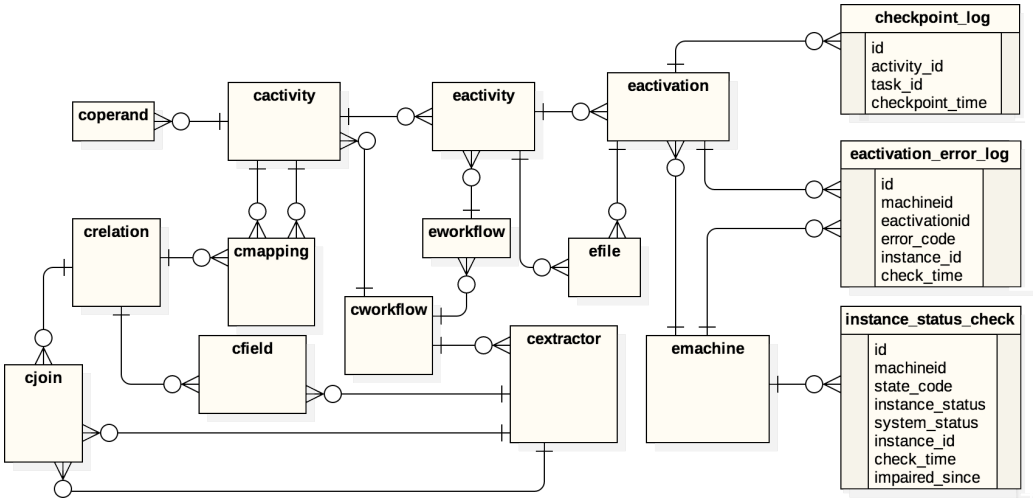

Figura 4.2: Modelo da base de proveniência estendido

Para solucionar a questão do monitoramento do ambiente computacional foi elaborado o Algoritmo 1, que, periodicamente, consulta informações[2] a respeito de cada uma das VMs instanciadas, disponibilizadas pelo provedor de computação em nuvem via API, e as armazena na base de dados que suporta o SGWfC.

Algoritmo 1 VerifyClusterHealth()

1: **while** *workflow* **is not finished do**
2: **for each** *machine* **in** *virtualCluster* **do**
3: $status = RequestStatus(machine)$
4: $Store(status)$
5: $Sleep(param)$

A tarefa de decidir se uma determinada máquina deve ou não ser classificada como problemática fica a cargo do Algoritmo 2. Ele toma informações históricas contidas na base de dados do SGWfC, que foram coletadas pelo Algoritmo 1 e pelo motor de execução do SGWfC, a respeito de falhas e sucessos ocorridos em execuções de tarefas em cada uma das máquinas. As informações sobre o *status* das máquinas, obtidos via provedor de nuvem, não deixam muito espaço para interpretação, *i.e* se foi informado que uma determinada máquina está com problemas, este problema precisará ser contornado. Entretanto, é possível que se trate de uma situação transiente. Desta forma, para evitar

[2]http://docs.aws.amazon.com/cli/latest/reference/ec2/describe-instance-status.html

reparos desnecessários, o Algoritmo 2 decide pela necessidade de reparo de uma determinada máquina somente após receber uma certa quantidade de avaliações negativas de *status* para a máquina em questão.

Utilizando-se o histórico de execuções realizadas pelo SGWfC, também é possível verificar se falhas ocorrem com uma frequência maior que dado limite percentual 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 problema. Apesar de serem armazenadas informações históricas a respeito das execuções e "status" das máquinas, são utilizados para efeito de avaliação somente dados referentes a execução corrente. Desta forma, evita-se que insucessos passados (e não mais relacionados à execução atual) influenciem na decisão. São utilizadas, portanto, somente dados relativos à execução corrente.

Algoritmo 2 SelectBadMachines()

1: $machines = List\emptyset$
2: **for each** *machine* **in** *virtualCluster* **do**
3: **if** *CountBadStatus(machine)* $>$ *param* **then**
4: *machines.add(machine)*
5: $x = CountTotalExecutions(machine) * perc$
6: $y = CountFailedExecutions(machine)$
7: **if** $x < y$ **then**
8: *machines.add(machine)*

Neste Algoritmo, as linhas 3, 5 e 6 dizem respeito a consultas SQL à base de dados que suporta a execução do SGWfC.

Por fim, foi proposto o algoritmo proativo chamado FTAdaptation (Algoritmo 3), que age como um monitor do SGWfC. Este algoritmo busca sanar possíveis fontes de falhas antes que estas ocorram efetivamente em execuções de *jobs* ou, ainda, reduzir estas ocorrências para que o impacto seja o menor possível. Ele faz uso dos dois algoritmos auxiliares (Algoritmo 1 e 2) exibidos anteriormente.

Algoritmo 3 FTAdaptation()

while *workflow is not finished* **do**
VerifyClusterHealth()
machineList = SelectBadMachines()
if *machineList is not empty* **then**
RequestMachinesByType(machineList)
RequestMachineTermination(machineList)
Sleep(t)

Este algoritmo é executado concorrentemente à execução do workflow científico. A

cada intervalo de tempo t (configurável através de um parâmetro) são coletadas informações sobre as instâncias e é refeito o cálculo da relação entre execuções com sucesso e sem sucesso para cada uma das máquinas. Caso sejam detectadas máquinas conforme os critérios de cada um dos algoritmos auxiliares, o Algoritmo 3 pausa a execução do workflow, realiza uma requisição de novas máquinas (do mesmo tipo que as máquinas problemáticas) e uma requisição de término das máquinas com problemas. Por fim, o *cluster* virtual é remontado e o workflow tem sua execução retomada. A parada do sistema para remontagem do *cluster* é necessária devido à implementação do SciCumulus, que utiliza MPI. Com isso, ocorre a perda de tempo de execução do trabalho que vinha sendo realizado corretamente, somente para a inclusão de novas máquinas. Tal abordagem tende a não ser viável conforme cresce o número de máquinas alocadas ao *cluster* virtual. Percebe-se, assim, a necessidade do estudo de alternativas ao MPI quanto à montagem dos ambientes distribuídos para execução dos workflows. Existe, ainda, a possibilidade de a máquina estar enfrentando problemas devido a falta de recursos para o processamento de determinada tarefa e, desta forma, a simples requisição de máquinas equivalentes não resolveria o problema. Entretanto, o esquema proposto pode ser estendido, incluindo um módulo capaz de decidir pelo melhor tipo de máquina a ser requisitado em cada um dos casos, semelhantemente ao proposto por Coutinho *et al.* [18].

Ao contrário da abordagem proposta neste trabalho, a maioria dos SGWfC são reativos, ou seja, se limitam a analisar o seu resultado (*Exit_Status* retornado pelo sistema operacional, por exemplo) após a execução de cada um dos *jobs*, e, ao detectar uma falha, simplesmente reexecutar o *job*.

A Figura 4.3 tem por objetivo ilustrar a execução de um workflow no SGWfC, trabalhando em conjunto com os módulos de tolerância a falhas propostos. Conforme pode ser visto neste esquema, existem 3 módulos principais e independentes. Enquanto o módulo principal do SGWfC (aqui simplesmente denotado por "Escalonador") realiza todo o processamento do workflow, desde a instanciação do cluster virtual e escalonamento de tarefas nas máquinas disponíveis até a coleta de resultados, o módulo Monitor fica responsável pelo monitoramento do ambiente, através do Algoritmo 1, e o módulo Adaptador realiza as intervenções conforme as necessidades. É interessante notar que a base de dados é acessada por todos os módulos do SGWfC, resultando em um ponto único passível de falhas capazes de impactar todo o sistema. Desta forma, esta base deve ser mantida em um recurso confiável. Apesar de ilustrada em um local fora da nuvem no esquema, esta base pode ser fornecida também pelos provedores de computação em nuvem, com certas garantias mínimas de qualidade de serviço. Outro ponto de falhas que deve ser notado é

o repositório de dados. Esta questão será abordada adiante.

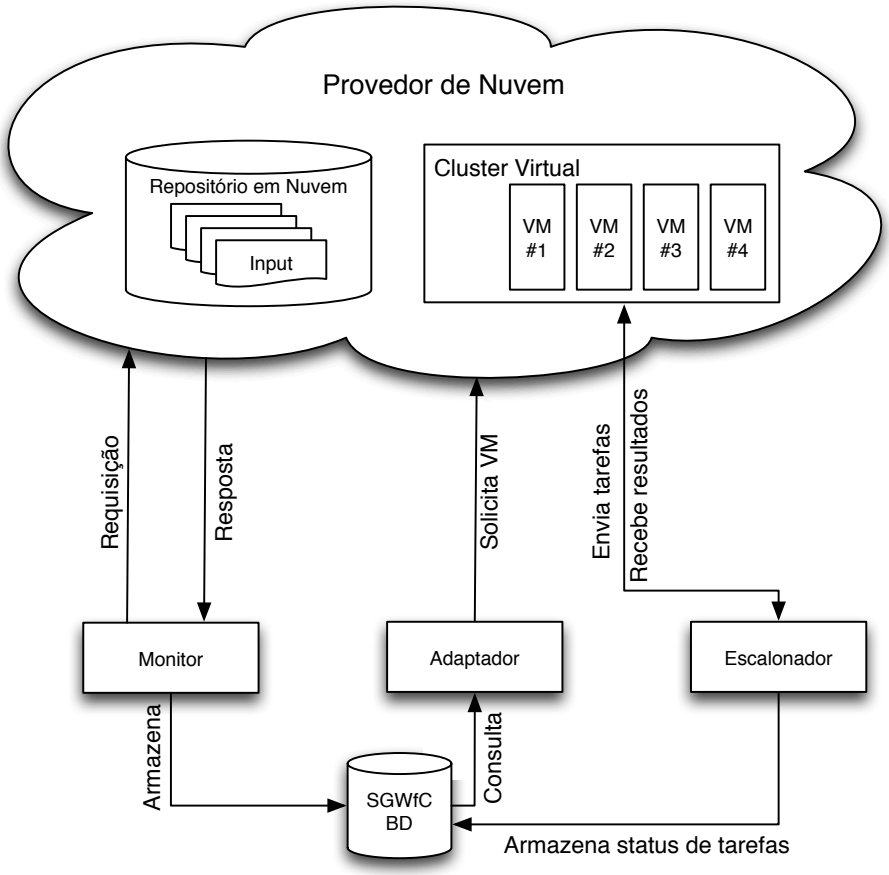

Figura 4.3: Modelo de execução de um workflow no SciCumulus

4.2 Tolerância a Falhas em Workflows Científicos

Conforme já mencionado, este trabalho tem com principal objetivo auxiliar os cientistas na execução de seus workflows baseados em nuvem, fornecendo uma solução para o manejo das diferentes falhas que ameaçam a execução de sistemas de larga escala em nuvem. Além disso, dadas as diversas possibilidades em termos de técnicas a serem utilizadas, este trabalho busca também auxiliar a decisão dentre as TTF disponíveis para cada uma das atividades a serem executadas. O estudo da literatura possibilita percebermos que Retentativa, Checkpoint-Restart e Replicação são as técnicas mais comumente utilizadas pelos SGWfC na tentativa de tornar a execução dos workflows capaz de se recuperar de falhas. Isto ocorre devido, principalmente, à facilidade de implementação e generalidade destas técnicas, ou seja, são passíveis de serem utilizadas com diferentes tarefas, sem que seja obrigatoriamente necessário reimplementar partes da mesma, e são capazes de

suportar diferentes classes de falhas.

As subseções a seguir trazem detalhes a respeito da implementação de cada uma destas técnicas no SGWfC SciCumulus[3] [21].

4.2.1 Retentativa

A técnica de Retentativa, ou Reexecução, consiste na execução de uma dada tarefa até que a mesma seja realizada com sucesso. Define-se um parâmetro máximo de tentativas para evitar *loops* infinitos em caso de problemas sem solução (erro na implementação do software, por exemplo). Especificamente no SciCumulus, uma vez que os dados de entrada e saída dos programas componentes do workflow são armazenados em um repositório centralizado e acessível por todas as máquinas, há a possibilidade de realização de retentativas em máquinas diferentes sem custos associados, ficando esta decisão somente a cargo do escalonador, que decide baseado na disponibilidade das máquinas e na fila de execução. Ocorrida uma falha, a tarefa tem seu contador de falhas incrementado, seu *status* atualizado de `RUNNING` para `READY` e volta para o fim da fila de atividades prontas para execução.

Esta técnica de tolerância a falhas é a mais básica e mais usualmente utilizada pelos SGWfC atuais devido à sua facilidade de implementação. Este trabalho busca fornecer novas ferramentas ao SGWfC para que as falhas possam ser tratadas da melhor forma possível e a técnica de Retentativa será utilizada para comparação de desempenho durante o restante do trabalho.

4.2.2 Checkpoint-Restart

A Figura 4.4 ilustra o SciCumulus executando um workflow em um cluster virtual em nuvem que consiste em 4 VMs. Cada uma destas VMs executa uma instância do SciCumulus responsável pela transferência de dados, execução de atividades e etc. A comunicação entre VMs é realizada através de troca de mensagens MPI. A Máquina Virtual 1 executa uma instância *master* do SciCumulus, responsável pelo escalonamento das atividades, enquanto os demais componentes são responsáveis por requisitar atividades para processar, atuando segundo um modelo Mestre/Trabalhador.

Para implementar C/R no SciCumulus foi necessário adicionar uma funcionalidade ao componente Mestre do SciCumulus, em execução na Máquina Virtual 1. Esta funci-

[3]http://scicumulusc2.wordpress.com/

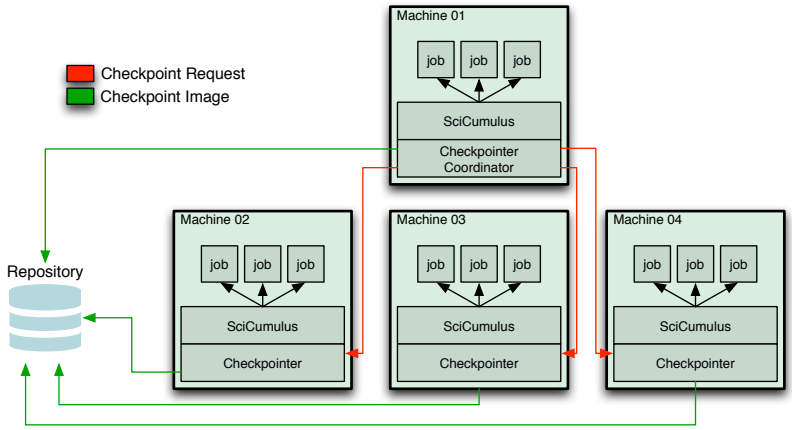

Figura 4.4: Arquitetura do *Checkpointer* Distribuído no SciCumulus

onalidade é descrita pelo Algoritmo 4, que determina quais ativações são elegíveis a um checkpoint. Uma ativação é considerada elegível à gravação de um checkpoint a partir do seu tempo de execução desde o último checkpoint. Esta informação é obtida através do banco de dados de proveniência mantido pelo SciCumulus, na tabela *checkpoint_log*, mencionada anteriormente. Desta forma, a partir de uma simples consulta SQL é possível determinar exatamente quais ativações deverão sofrer um checkpoint. Para cada atividade i cujo tempo de execução desde o último checkpoint t_i é maior que δ, onde δ é um parâmetro de entrada, uma requisição de checkpoint é enviada para a instância SCCore responsável pela execução da atividade$_i$. Cada instância do SCCore executa o Algoritmo 5, aguardando por requisições de checkpoint e, ao recebê-las, executa o Algoritmo 6.

Algoritmo 4 CheckpointCoordinator()

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

Algoritmo 5 CheckpointListener()

1: **while** *workflow* **is not finished do**
2: *Receive*(*msg*)
3: *CallCheckpoint*(*msg.workspace*)

Uma requisição de checkpoint contém o *workspace* da atividade - um diretório no sistema de arquivos - onde a atividade está sendo executada. Dado que cada ativação está associada a um *workspace* único, é trivial obter o ID do processo associado a esta ativação através de chamadas de SO. A seguir, o processo em questão é pausado, sua imagem (pilha

Algoritmo 6 Checkpoint(*proc*)

1: *pid = pgrep(proc)* * *Consulta o PID do processo* *
2: *image = Dump(pid)* * *Pausa o processo e armazena sua imagem em disco* *
3: *ws = Compress(proc.workspace)* * *Comprime workspace do processo* *
4: *kill -SIGCONT pid* * *Resume o processo* *
5: *Compress(image)* * *Comprime a imagem do processo* *
6: *Move(ws)* * *Move arquivos para repositório* *
7: *Move(image)* * *Move arquivos para repositório* *
8: *DeleteLastCheckpoint()* * *Apaga checkpoint antigo* *

de execução e memória) é armazenada em disco através da ferramenta CRIU[4], arquivos de entrada e saída parcial são copiados e incluídos aos arquivos de imagem do processo e este conjunto de dados é movido para um repositório seguro e escalável.Em seguida, são inseridas entradas na tabela *checkpoint_ log* contendo a identificação da ativação a que este checkpoint se refere, seu *timestamp* e local onde foi armazenado e, finalmente, o processo é reiniciado. Neste trabalho foi utilizado como repositório o Amazon S3[5]. Segurança e escalabilidade são duas questões cruciais no tratamento do checkpoint armazenado pois, ao ser requisitado, o mesmo deverá ser facilmente acessível e íntegro. A escalabilidade deve ser considerada pois tal repositório deve estar preparado para receber numerosas requisições de escrita em um curto espaço de tempo. Caso estas questões não sejam resolvidas, há o risco de criar-se um gargalo no sistema, degradando o desempenho do checkpointer. O uso do serviço de armazenamento Amazon S3, à exemplo de diversos outros serviços como o Google Cloud Storage[6] e MS Azure Storage[7] por exemplo, fornece uma fácil solução para estas questões: tal serviço realiza automaticamente replicação de dados em diferentes localidades, além de suportar leitura e escrita em paralelo na ordem de centenas de requisições por segundo[8].

Dada a ocorrência de uma falha em uma ativação qualquer, o escalonador do SGWfC tenta executar esta ativação novamente (técnica *default* de Reexecução). Entretanto, caso exista uma imagem de checkpoint para esta atividade, em vez de executá-la desde o início, o escalonador busca a imagem do checkpoint encontrada no repositório e a extrai no workspace da atividade, sendo assim possível reiniciar a ativação a partir do ponto onde a execução se encontrava no momento do último checkpoint gravado. O Algoritmo 7 ilustra este procedimento. C-R é também aplicável a cenários onde uma ativação esteja em execução por um período de tempo muito longo em uma determinada VM. Detectada

[4]http://criu.org/
[5]https://aws.amazon.com/s3/
[6]https://cloud.google.com/storage/
[7]https://azure.microsoft.com/en-us/services/storage/
[8]http://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html

esta situação, o escalonador pode decidir migrar esta ativação para uma nova VM com mais poder de processamento e, em vez de ter de realizar todo o processamento novamente, pode utilizar o trabalho parcialmente realizado na máquina anterior.

Algoritmo 7 Restart(*proc*)

1: *ckpt = Copy(proc.files)*
2: *Extract(ckpt, procImg)*
3: *Restart(procImg)*

Os Algoritmos 6 e 7 são baseados em ferramentas externas de Checkpoint-Restart. Este trabalho adotou o CRIU para a realização desta funcionalidade devido a vantagens em comparação com soluções similares como por exemplo o BLCR[9]. Suas vantagens incluem: (i) não necessidade de se carregar bibliotecas na aplicação que realizará a tarefa em questão, (ii) obtenção de C-R sem a necessidade de alteração do código-fonte dos programas e (iii) o armazenamento de arquivos abertos. Estas características são importantes no contexto de workflows científicos uma vez que eles - normalmente - englobam a execução de diversos programas fornecidos por terceiros, que muitas vezes são caixa-preta e estaticamente ligados.

4.2.3 Replicação de Tarefas

Além da solução de Checkpoint-Restart, este trabalho implementou mecanismos de replicação de tarefas no SGWfC SciCumulus. Replicação, conforme visto anteriormente, é a técnica na qual uma unidade de trabalho é copiada ao menos uma vez, possibilitando algum grau de resiliência devido à redundância [32]. Desta forma, em um sistema cujo esquema de replicação executa N cópias de uma mesma tarefa ao mesmo tempo, é possível tolerar até N-1 falhas pois basta que uma das cópias termine sua execução com sucesso. Assim, a execução do workflow pode prosseguir utilizando-se a saída de qualquer uma das tarefas terminadas com sucesso.

O esquema de replicação aqui desenvolvido foi construído a partir do modelo de execução de workflows baseado em álgebra relacional proposto por Ogasawara *et al.* [55]. Segundo este modelo, dados são representados por relações e as *atividades* são mapeadas em diferentes operadores, conforme seu padrão de consumo e produção de dados. A Figura 4.5 representa este esquema. Nesta figura temos a relação A, composta por um conjunto de tuplas, sofrendo uma transformação definida pela operação OP, do tipo `MAP` (N tuplas de entrada geram N tuplas de saída), e gerando como resultado a relação B. A operação

[9] http://crd.lbl.gov

OP, que representa um programa definido durante a composição do workflow, processa cada uma das tuplas contidas na relação A e gera as tuplas que formarão a relação B. Ao conjunto formado pela operação OP e as tuplas de entrada necessárias para uma única execução do programa que define a operação OP, dá-se o nome de *ativação*.

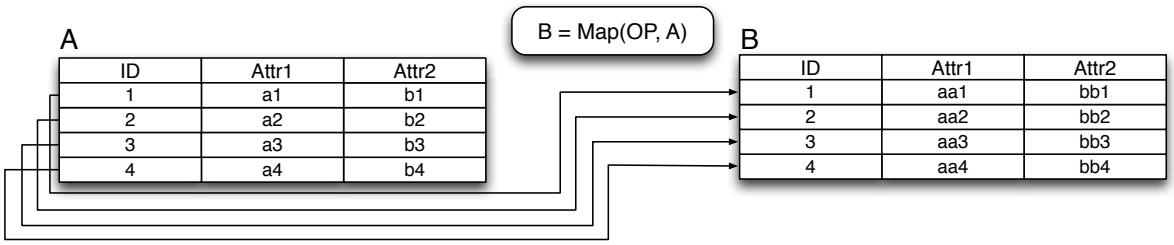

Figura 4.5: Modelo de execução baseado em algebra relacional

O esquema de replicação proposto neste trabalho estende cada uma das relações de entrada do SciCumulus, incluindo alguns atributos. A inclusão destes atributos tem por objetivo (i) possibilitar a identificação das unidades de trabalho, ou seja, das ativações, e (ii) fornecer um meio de ordenação entre tarefas e cópias. O objetivo (i) é alcançado através do atributo aqui denominado `GROUP_ID`. A cópias de uma mesma tupla de entrada são atribuídos o mesmo valor de `GROUP_ID`. Desta forma, a conclusão de uma execução com sucesso neste esquema de replicação passa a ser denotada pela existência de ao menos uma tupla de saída para cada grupo de tuplas de mesmo `GROUP_ID`. Vale frisar que esta condição abrange também os casos onde não se deseja realizar a replicação das tarefas, sendo desta forma cada grupo composto por apenas 1 tupla. A Figura 4.6 representa este esquema de replicação.

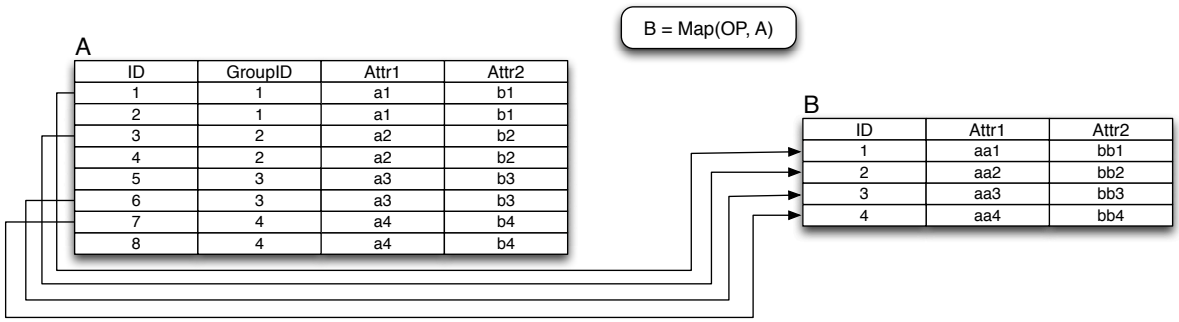

Figura 4.6: Replicação baseada em álgebra relacional

Neste exemplo, foi realizada a replicação de ativações através da duplicação das tuplas de entrada. O escalonador fica livre para decidir qual o melhor critério de escalonamento

de forma transparente.

O objetivo (ii) é alcançado através de um atributo aqui chamado `PRIORITY`. O uso deste atributo possibilita a atribuição de prioridades iguais ou diferentes para cópias e suas tarefas origem, *i.e.* suas cópias primárias. Com isso, é possível projetar uma execução mais conservadora, ou seja, mais preparada para lidar com falhas ao escalonar tarefas primárias e suas cópias ao mesmo tempo ou, ainda, tomar uma atitude menos conservadora, dando prioridade a tarefas primárias e deixando as cópias em espera por recursos livres.

Exemplo:

- Ambiente computacional contendo 2 processadores
- 2 tarefas a serem executadas
- esquema de replicação cria 1 cópia de cada tarefa

Em um sistema sem replicação, cada tarefa é executada por algum dos processadores e, em caso de falhas, alguma providência deverá ser tomada (a reinicialização do processamento, por exemplo). Caso seja utilizado o esquema de replicação com prioridades iguais para tarefas primárias e suas cópias, teremos uma tarefa (juntamente com sua cópia) em execução enquanto a próxima aguarda junto com sua cópia. Caso ocorra uma falha, basta o resultado da cópia remanescente para que o processamento possa continuar sem problemas. Entretanto, caso não ocorram falhas, tem-se o desperdício de recursos que poderiam ter sido aplicados na execução da tarefa em espera. Portanto, o objetivo do parâmetro `PRIORITY` é fornecer ao usuário do SGWfC flexibilidade suficiente para decidir qual atitude tomar baseado, por exemplo, em informações como a quantidade de recursos disponíveis (criação de réplicas levará ao enfileiramento de tarefas?), a probabilidade de falha da tarefa (o histórico desta tarefa mostra alta taxa de falhas?) ou ainda a extensão de uma falha em tal tarefa (uma reinicialização desta tarefa levará ao aumento do makespan do workflow?).

Este último cenário é interessante em situações de baixo risco de ocorrência de falhas ou para tarefas de curta duração (baixo custo de reinicialização). Entretanto, dado que o escalonamento de cópias após a ocorrência de falhas em tarefas primárias é um caso equivalente à utilização da técnica de retentativa, o foco deste trabalho será na utilização de réplicas com a mesma prioridade que suas tarefas primárias.

Para coordenar a execução dos workflows neste esquema de replicação foram propostos

dois algoritmos, Algoritmo 8 e Algoritmo 9. Eles têm por objetivo manter o workflow consistente, além de evitar o desperdício de recursos.

O Algoritmo 8 é executado em cada instância Trabalhadora do SCCore. Ele envia requisições ao nó Mestre e executa as tarefas que lhe são enviadas. Este esquema foi estendido para enviar notificações ao fim do processamento de uma tarefa com sucesso. Estas notificações são enviadas ao nó Mestre, informando tal situação através do método `Notify`. Este método envia uma mensagem ao nó Mestre contendo o identificador da atividade em questão e o `GROUP_ID` da ativação.

Algoritmo 8 ExecuteActivation()

1: *$act = RequestActivation()*
2: *Execute($act)*
3: **if** *$act.finish_status == SUCCESS* **then**
4: *Notify($m, $act)*
5: **else**
6: **if** *$act.finish_status == FINISHED_REP* **then**
7: *FinishNormally()*

Ao receber uma notificação de término de tarefas, o nó Mestre, que executa o Algoritmo 9 como um *listener*, consulta na base de dados do SGWfC quais outras máquinas estão processando ativações de `GROUP_ID` e *ACTIVITY_ID* iguais aos recém recebidos e envia mensagens a estas máquinas (instâncias Trabalhadoras do SCCore) para que estas também forcem o encerramento da execução, uma vez que uma cópia já foi concluída com sucesso. Estas cópias são marcadas com um *status* criado especialmente para este caso, denominado `FINISHED_REP`, indicando que a tarefa foi finalizada sem erros, mas seu resultado não é aproveitável.

Desta forma, o critério de término com sucesso foi atualizado. No lugar de buscar com que todas as ativações tivessem suas execuções completadas com sucesso, busca-se que cada grupo de N ativações, denotado por mesmo `GROUP_ID`, tenha ao menos uma ativação terminada com sucesso, tolerando assim N-1 falhas.

Algoritmo 9 WaitForNotifications()

1: *$msg = ReceiveNotification()*
2: *cancelledGroups.add($msg.group_id)*
3: *$machines = Query($msg.act.id, $msg.act.group_id)*
4: **for each** *$machine* **in** *$machines* **do**
5: *StopActivations($m.PWS, $m.rank)*

Capítulo 5

Resultados Experimentais

Esta seção apresenta os resultados e análises dos experimentos realizados sobre as técnicas de tolerância a falhas desenvolvidas neste trabalho e implementadas no SGWfC SciCumulus. Para o estudo, foram criados casos de teste utilizando dois workflows distintos - SciPhy e Montage - de forma a explorar as diferentes características de tarefas realizadas por cada um destes workflows. Além disso, estes workflows foram executados em diferentes configurações de ambientes computacionais e submetidos a diferentes padrões de falhas simuladas. Com isso, é possível verificar os pontos fortes e fracos de cada uma das abordagens de tolerância a falhas propostas. A comparação das técnicas elaboradas neste trabalho com a técnica Retentativa permite a observação do *overhead* introduzido por cada uma delas, já que a técnica de Retentativa não pressupõe qualquer gasto adicional que possa vir a afetar o desempenho da aplicação em execução. Nos casos com falhas, são simuladas falhas ao fim de todas as ativações da atividade em questão.

5.1 Workflow Montage

Montage [8] é um workflow de astronomia que coleta dados de imagens do espaço, captadas por telescópios, e cria mosaicos através da justaposição destas imagens, sendo possível retratar diversos objetos de interesse astronômico. A composição deste workflow pode ser dada de diferentes maneiras e, de acordo com o interesse do cientista, etapas podem ser adicionadas ou removidas. A Figura 5.1 apresenta uma possível *composição* para este workflow, consistindo de 9 atividades. Cada atividade representa uma etapa de processamento que poderá ser realizada uma única vez ou repetidas vezes. A representação é interessante pois mostra como cada tarefa se comporta em termos de entrada e saída de dados. Ainda de acordo com a abordagem proposta por Ogasawara *et al.* [55], atividades

podem ser do tipo `MAP` (1 entrada gera 1 saída), `SPLIT_MAP` (1 entrada gera N saídas), `REDUCE` (N entradas geram 1 saída), `FILTER` (N entradas geram M<N segundo critério de filtro), etc. O tipo da atividade indica também se haverá a possibilidade de execução em paralelo de muitas ativações. Estas informações podem ser utilizadas na decisão de se escalonar réplicas para as ativações ou não.

Este workflow foi executado repetidas vezes de forma a coletar dados de execução para as análises que serão realizadas neste trabalho. Erros foram induzidos - foi enviado sinal *SIGKILL* para processo - em cada uma das atividades, pois suas peculiaridades influenciam em como as mesmas se recuperam das falhas ou, ainda, na dificuldade de torná-las tolerante a falhas. As Figuras 5.2, 5.3 e 5.4 mostram os tempos de execução de cada uma das atividades do workflow Montage utilizando-se cada uma das técnicas de tolerância a falhas desenvolvidas neste trabalho (Retry, C-R e Replicação), tanto na presença de falhas quanto em situações livres de falhas (NF - No Fault), em 3 diferentes configurações de ambientes computacionais (4, 16 e 22 CPUs). Foi utilizado intervalo entre checkpoints de 30 segundos e fator replicação de tarefas 2 (cada tarefa tem uma única cópia, com prioridade igual à tarefa primária).

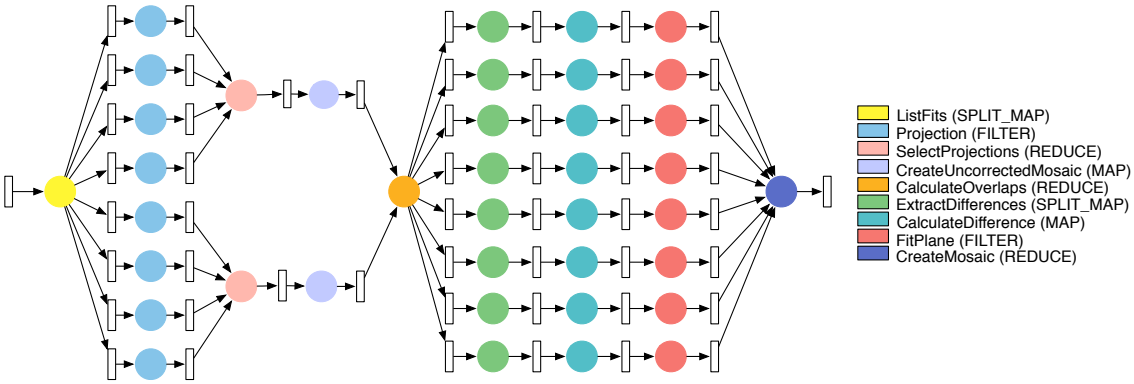

Figura 5.1: Workflow Montage

A maioria das atividades executadas são consideradas de curta duração pois levam tempo inferior a 20 segundos para serem processadas (utilizando as TTF Retry ou Checkpoint-Restart). A única exceção é a atividade *Projection*. Dentre todas estas atividades de curta duração, é interessante notar duas em especial, devido à grande perda de desempenho observada ao utilizar a técnica de Replicação: `CalculateDifferences` e `FitPlane`. A Tabela 5.1 ajuda a elucidar os motivos por trás de tal degradação. É interessante notar que estas duas atividades são responsáveis pelo processamento dos maiores *datasets*. Desta forma, a replicação destas atividades - com a consequente criação do dobro de ativações - resulta em um grande aumento na demanda por processamento, com

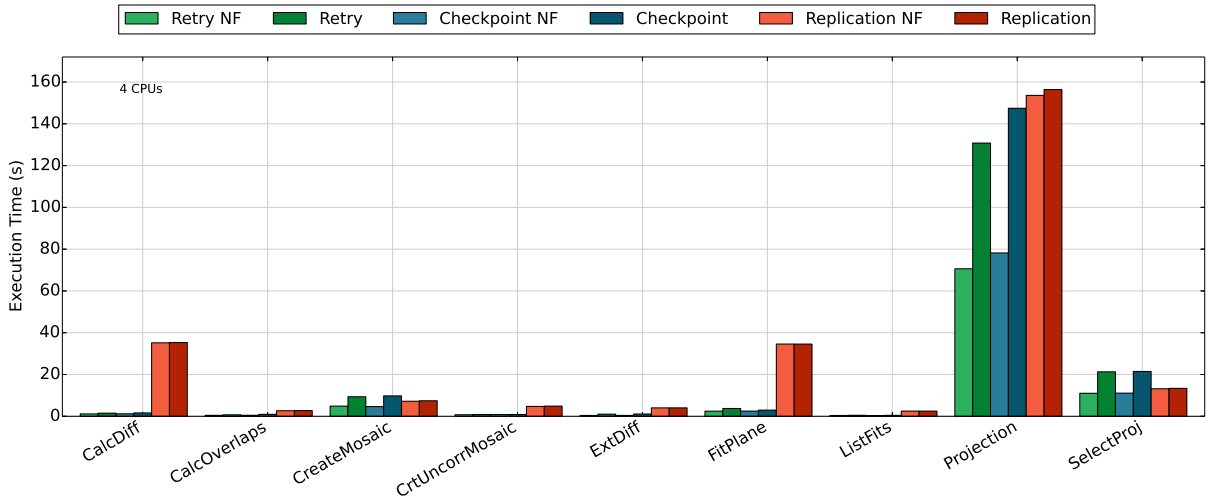

Figura 5.2: Desempenho das TTF para o workflow Montage com 9 atividades e 4 cpus

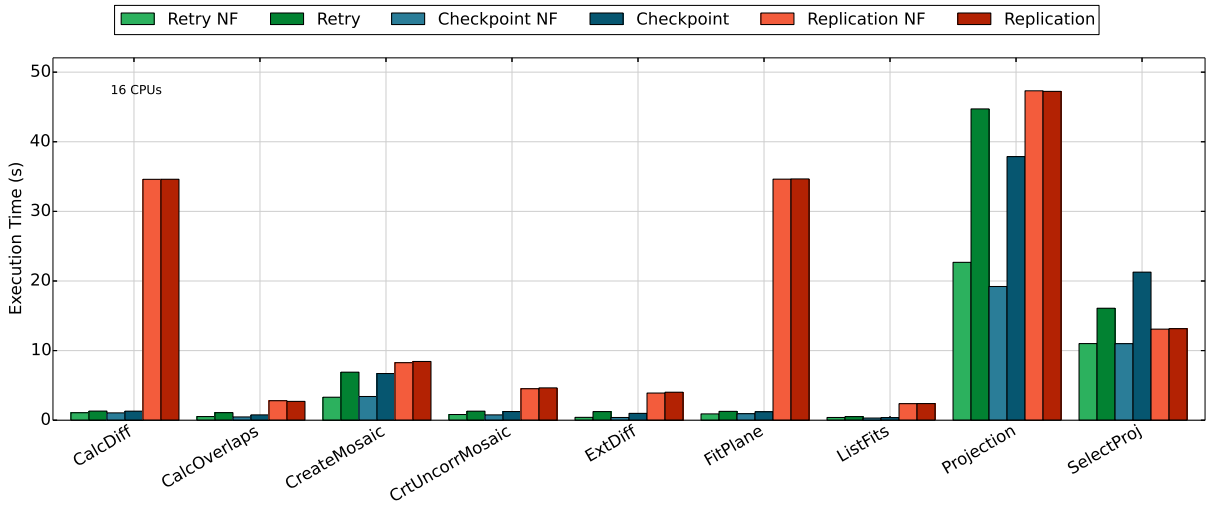

Figura 5.3: Desempenho das TTF para o workflow Montage com 9 atividades e 16 cpus

um grande número de ativações prontas em paralelo, resultando em aumento no tempo de fila, gastos com o gerenciamento desta fila, além de gastos com a correta terminação das réplicas redundantes. Como consequência, observa-se a perda de desempenho ilustrada pelos gráficos.

Por outro lado, atividades que geram poucas ativações têm desempenho similar para todas as 3 TTF comparadas. Conforme já mencionado, a replicação de atividades exige um trabalho adicional para o correto término das cópias. Se a atividade é composta de diversas tarefas de curta duração, é possível que este trabalho adicional seja maior do que o custo da tarefa em si. Isto pode ser visto ao comparar as atividades `CreateMosaic` e `SelectProjection` com quaisquer outras atividades de curta duração. Pode-se obser-

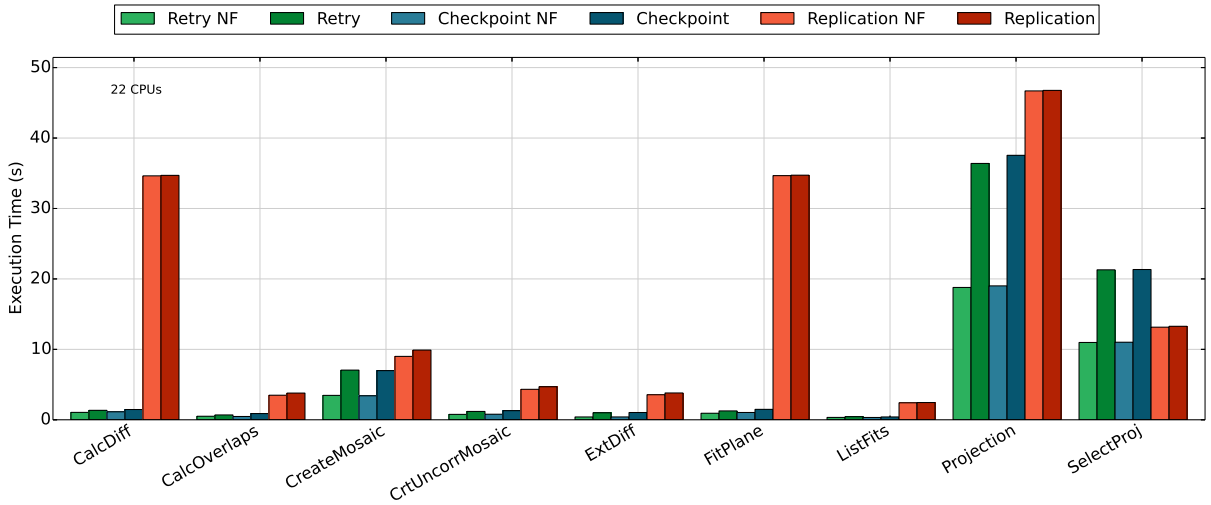

Figura 5.4: Desempenho das TTF para o workflow Montage com 9 atividades e 22 cpus

Ativ.	# de arq. processados	Avg. Re-tentativa	Avg. Replicação
CalculateDifference	17	0:01.121	0:34.808
CalculateOverlaps	1	0:00.523	0:02.995
CreateMosaic	1	0:04.266	0:08.206
CreateUncorrectedMosaic	1	0:00.809	0:04.549
ExtractDifferences	1	0:00.466	0:03.842
FitPlane	17	0:01.609	0:34.628
ListFits	1	0:00.392	0:02.427
Projection	10	0:45.918	1:22.152
SelectProjections	1	0:11.954	0:13.153

Tabela 5.1: Tempos de execução das atividades do workflow Montage para diferentes TTF

var a partir da Figura 5.2, com 4 CPUs, que a técnica de Replicação tem desempenho semelhante em cenários livres de falhas e melhor que as outras duas TTF em cenários com falhas. Conforme aumenta a quantidade de recursos disponibilizada e atividades executam mais rápido, o desempenho da técnica Retry melhora enquanto a Replicação tem piora no seu desempenho para a atividade `CreateMosaic`. Quanto à atividade `SelectProjections`, Replicação ainda tem o melhor desempenho entre as demais.

É importante explicitar que nenhuma atividade foi capaz de restaurar um checkpoint durante este experimento uma vez que todas as tarefas executadas neste workflow são terminadas em menos de 30 segundos (parâmetro definido para intervalo entre checkpoints). Assim, ainda que a técnica Checkpoint-Restart tenha sido definida para o experimento, como não havia qualquer checkpoint disponível no momento da restauração, a recuperação foi realizada através de uma Reexecução. Por este motivo, os resultados das duas

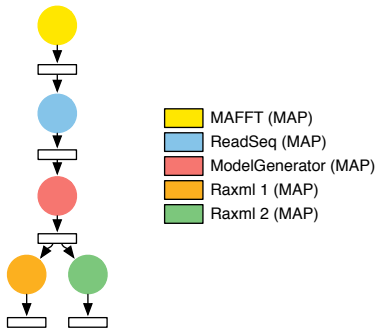

Figura 5.5: Workflow SciPhy

técnicas são basicamente os mesmos neste experimento.

5.2 SciPhy Workflow

SciPhy é um workflow do domínio da Biologia que recebe um *dataset* de sequências de DNA, RNA e aminoácidos e cria modelos evolucionários com o objetivo de se determinar ancestrais em comum para grupos de organismos [54]. Conforme pode ser visto na Figura 5.5, este workflow consiste basicamente de um *pipeline* contendo 3 atividades sequenciais seguidas por 2 atividades que podem ser executadas em paralelo. Este experimento foi realizado de forma similar ao apresentado anteriormente para o workflow Montage. Foram realizadas repetidas execuções, porém, diferentemente do experimento anterior, variou-se o número de arquivos de entrada (cada arquivo de entrada contém genomas inteiros e dá origem a um *pipeline* SciPhy). Portanto, as Figuras 5.6, 5.7, 5.8 representam o tempo médio de execução de cada atividade do workflow SciPhy utilizando-se como entrada, respectivamente, 1, 4 e 8 genomas, sendo executados em 3 ambientes computacionais diferentes (ver Tabela 5.2), com exceção para o caso com 8 entradas que não foi executado no Ambiente 1 pois o tempo de execução aumentaria consideravelmente devido ao enfileiramento das ativações.

	Nº de CPUs	Memória RAM
Ambiente 1	2	4 GB
Ambiente 2	4	8 GB
Ambiente 3	8	16 GB

Tabela 5.2: Ambientes computacionais utilizados para execução do SciPhy

Semelhantemente ao experimento do Montage, a maioria das atividades não gasta muito tempo em processamento, com exceção para as atividades `ModelGenerator` e `RAxML`.

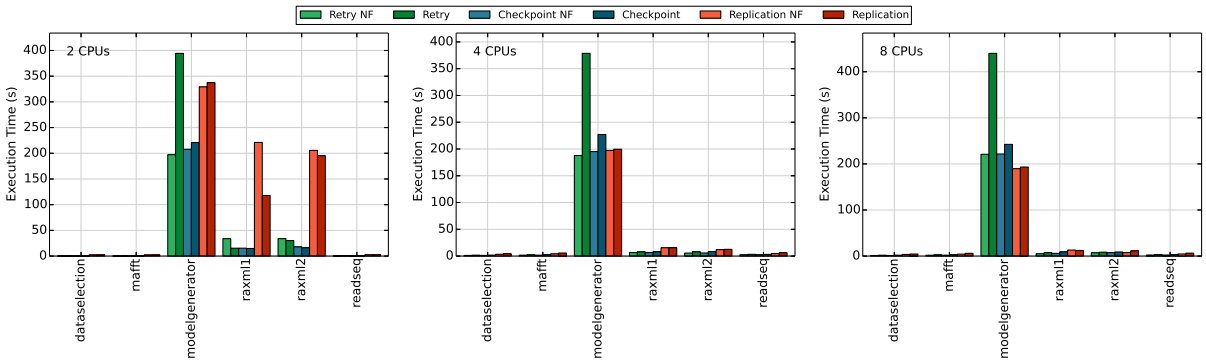

Figura 5.6: Desempenho das TTF para o workflow SciPhy com 1 genoma de entrada

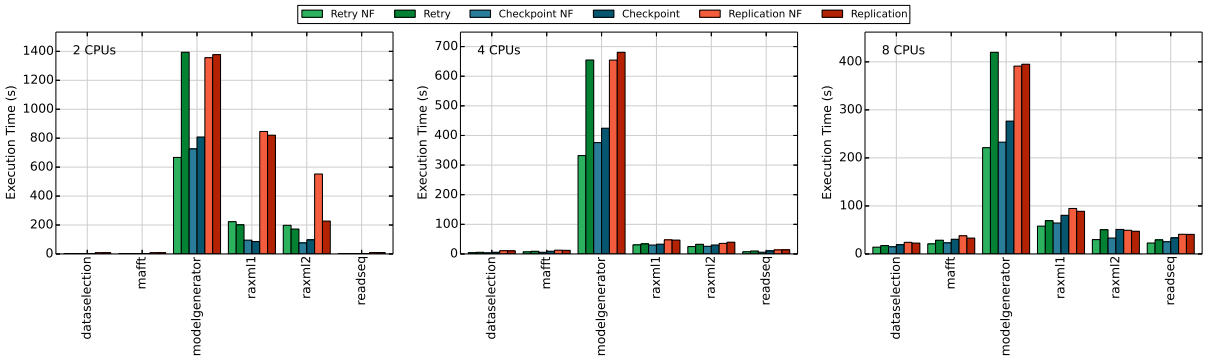

Figura 5.7: Desempenho das TTF para o workflow SciPhy com 4 genomas de entrada

Como o `RAxML` utiliza processamento em paralelo, a execução concorrente destas tarefas deteriora o desempenho de forma considerável por conta da degradação de cache [61]. `RAxML` foi configurado para utilizar 2 *threads*, o menor valor possível para esta aplicação.

Ao analisar as Figuras 5.6, 5.7 e 5.8, é possível observar o efeito do aumento do número de genomas de entrada em relação ao comportamento das TTF. A Figura 5.7 mostra que a técnica de Replicação não mais é vantajosa para nenhum dos ambientes computacionais considerados. Percebe-se que o desempenho nesta situação, para ambos cenários de melhor caso e pior caso são comparáveis ao pior caso utilizando a técnica Retry. C/R tem o melhor desempenho para a atividade `ModelGenerator` nas 3 configurações de máquina virtual. Vale frisar que a técnica de Checkpoint desenvolvida neste trabalho não pode ser executada em paralelo (gravação de mais de um checkpoint ao mesmo tempo) devido a limitações na ferramenta CRIU. Entretanto, uma vez que a solução possui um *overhead* significativamente baixo, isto não tem grande impacto nos resultados, conforme pode ser verificado nas Figuras 5.7 e 5.8.

Ao comparar os resultados da atividade `ModelGenerator` utilizando as técnicas Re-

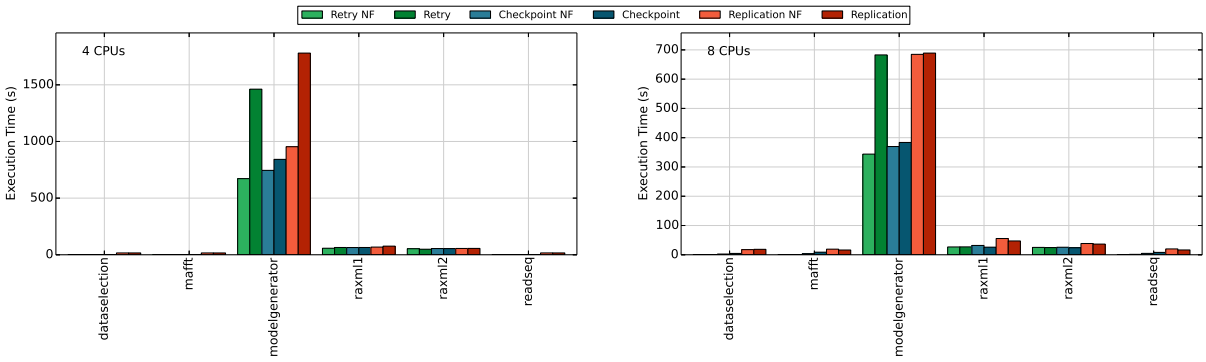

Figura 5.8: Desempenho das TTF para o workflow SciPhy com 8 genomas de entrada

tentativa, Checkpoint-Restart e Replicação nas diferentes máquinas virtuais em seus *best case cenarios* (sem falhas), é possível notar que a técnica de Checkpoint traz uma leve degradação no desempenho, de cerca de 4%, utilizando-se um parâmetro de intervalo entre checkpoints de 30 segundos. A utilização da técnica de Replicação, por outro lado, quase sempre resulta em tempos de execução cerca de 100% maiores, em comparação com a Re-tentativa. Isto é observado pois, enquanto a técnica Re-tentativa não necessita a realização de nenhum trabalho extra, (i) a técnica de Checkpoint requer troca de mensagens, monitoramento de processos em execução, além de, periodicamente, sua pausa e reinicialização para armazenamento das imagens de checkpoint e (ii) a técnica de Replicação realiza o dobro do trabalho de processamento. Dado que o ambiente disponível para cada um dos experimentos é somente suficiente para o processamento de cada uma das ativações (já que o número de entradas é igual ao número de CPUs), temos que a replicação causa o enfileiramento das ativações. Entretanto, há casos onde o uso de ambas as técnicas se justifica. O uso desta técnica de *checkpoint-restart* é justificado pela garantia de um limite superior no tempo perdido em caso de falhas - o intervalo entre checkpoints. Este benefício é obtido com a contrapartida de um custo razoavelmente baixo, enquanto outras técnicas não são capazes de entregar tal desempenho em termos de limite de perdas (Re-tentativa) ou *overhead* (Replicação). Já a técnica de replicação tem sua utilização suportada pelos resultados observados nos experimentos com 1 genoma de entrada, com 4 e 8 CPUs (Figura 5.2), onde se observam melhores resultados em casos de falhas.

5.3 Modelo Preditivo para TTF em Workflows Científicos

Apesar de a análise realizada prover uma visão interessante a respeito da melhor TTF a ser utilizada em cada atividade em ambos os workflows, uma análise em larga escala possibilitaria decisões mais acertadas. Ao considerarmos a grande quantidade de dados coletados a partir de execuções prévias dos diferentes workflows, em diferentes configurações de ambientes e submetidos a diferentes padrões de falhas, é possível realizarmos a extração de padrões capazes de ajudar na escolha pela melhor técnica para uma determinada atividade de um workflow em um determinado ambiente computacional. Entretanto, a análise manual de tal volume de dados se torna tediosa e sujeita a erros, ou até inviável.

Com isso, este trabalho propõe a solução deste problema através de modelos preditivos utilizando-se algoritmos de aprendizado de máquina, como Árvores de Classificação e Indução CN2 [16, 59], por exemplo. Estes algoritmos foram treinados utilizando a base de dados coletados durante a execução dos experimentos acima mencionados, 13.628 traces de execução de ativações do workflows Montage e SciPhy. O objetivo de ambos os algoritmos é a criação de um modelo de previsão de um valor alvo baseado em um conjunto de variáveis de entrada. Estas variáveis de entrada são os diferentes parâmetros coletados durante a execução das tarefas dos workflows: nome da atividade, tipo da atividade (segundo a abordagem algébrica proposta por Ogasawara *et al.* [55]), o tipo de TTF utilizada, o número de tarefas paralelas associadas com a atividade (se *threads* foram utilizadas), o número de réplicas concorrentes (no caso de utilização da técnica de replicação) e o intervalo entre *checkpoints* (caso a técnica escolhida tenha sido *checkpoint-restart*). A variável alvo é o tempo de execução da atividade, já que estamos procurando a TTF que resulta em um tempo de execução mais baixo. O tempo de execução, neste caso, foi escolhido como parâmetro de comparação pois considera-se que todas as TTF disponíveis sejam capazes de solucionar as diferentes falhas que possam vir a ocorrer.

Foi utilizado o framework Orange Data Mining [1] para treinar os modelos com a base de entrada. Foram executados os seguintes passos:

1. Análise estatística dos atributos
2. Discretização da variável alvo (Discretization)
3. Amostragem de Dados (Data Sample)

[1] https://orange.biolab.si/

4. Treinamento da Árvore de Classificação e do Indutor CN2

5. Teste dos modelos (Test Learners)

Na etapa de Análise Estatística dos Atributos (1), foi verificada a distribuição de dados para cada um dos atributos. Antes de gerar a árvore de classificação e as regras com CN2, foram avaliadas estatísticas a respeito de cada um dos atributos utilizados no modelo. Esta análise revelou que a maior parte das falhas presente na base de dados ocorreu em atividades que se enquadram no tipo `MAP` (um input gera um output). Por outro lado, não foram encontradas falhas em atividades do tipo `SPLIT_MAP` (um input gera N>1 outputs). Este tipo de observação se deve ao fato de estas representarem um número muito pequeno de atividades e, ainda, que não são intensivas computacionalmente, sendo executadas rapidamente. Desta forma, falhas em tais atividades são menos frequentes.

O passo de discretização é necessário pois a variável alvo é o tempo de execução, um valor contínuo, e o objetivo não é determinar valores exatos de tempos de execução mas, (i) se uma dada atividade se classificaria como sendo de longa ou curta duração e (ii) se a utilização de diferentes TTF impactaria nesta classificação, justificando assim a escolha por determinada técnica. Foi realizada uma discretização *equal-frequency* que produziu 4 categorias de atividades de acordo com seus tempos de execução: `SHORT-TERM`, `MEDIUM-TERM`, `LONG-TERM` e `XLONG-TERM`.

Uma forma comum de organizar os dados para treinamento dos classificadores é dividir a base de treinamento em dois conjuntos, denominados conjuntos de treinamento e teste. O modelo preditivo é treinado utilizando-se o conjunto de dados de treinamento e seu desempenho é avaliado em termos de previsões de instâncias do conjunto de teste. Esta divisão é feita para que seja possível avaliar o desempenho do modelo produzido em situações de processamento de novas atividades em diferentes ambientes computacionais. Assim, o passo (3), Data Sample, tem por objetivo dividir a base de treinamento utilizando uma proporção tradicional: 70% de dados para a base de treinamento e 30% de dados para a base de aprendizado. A base de treinamento obtida foi, por fim, utilizada nos algoritmos de aprendizado de máquina considerados (passo 4). Ambos os modelos preditivos produzem um conjunto de regras que podem ser utilizados pelos SGWfC para determinar qual a melhor TTF para uma determinada atividade em um determinado ambiente computacional. Por exemplo:

```
IF activity_name=sciphy.dataselection AND
    ft_type=retry AND
```

```
    n_tasks=>7.00
THEN D_class=SHORT-TERM

IF activity_name=sciphy.dataselection AND
    ft_type=replication AND
    n_tasks=>7.00
THEN D_class=MEDIUM-TERM
```

Esta regra determina que caso a técnica *Retry* seja escolhida para a atividade `dataselection` no workflow SciPhy, sendo esta atividade composta por mais de 7 ativações, o tempo total de execução será classificado como `SHORT-TERM`. Por outro lado, caso a técnica escolhida seja Replicação, para o mesmo caso, o tempo de execução é classificado como `MEDIUM-TERM`, um cenário pior. Através da comparação da classificação do tempo da execução para uma determinada TTF escolhida, é possível determinar qual delas resulta em melhor desempenho e, assim, tomar a decisão. Uma vantagem dos algoritmos de árvore de classificação e CN2 é que ambos geram regras que podem ser facilmente compreendidas e, com isso, implementadas no SGWfC. Apesar das regras geradas parecerem de qualidade, é necessário avaliá-las utilizando-se a base de teste obtida no passo 3. Esta avaliação é realizada no passo 5, e a Tabela 5.3 apresenta os resultados obtidos.

Algorithm	Precision	Recall	F-Measure
Classification Tree	0.8502	0.8842	0.8669
CN2	0.8510	0.8779	0.8642

Tabela 5.3: Precisão dos modelos preditivos propostos

A Tabela 5.3 apresenta as médias (média de cada um dos valores da variável alvo) das métricas *Precision*, *Recall* e *F-Measure* para os modelos de árvore de classificação e CN2. Percebe-se que ambos os modelos apresentam valores maiores que 80% para todas as métricas, o que é um resultado promissor. Alto valor de *precision* significa que os algoritmos foram capazes de classificar corretamente a maior parte das instâncias da base de teste, enquanto um alto *recall* mostra que os classificadores foram capazes induzir corretamente uma parte relevante de cada um dos valores alvo. Por exemplo: se o classificador prevê somente um caso `SHORT-TERM` e ele está correto, temos *precision* de 100%. Entretanto, caso o universo de casos `SHORT-TERM` compreenda 10 casos, temos *recall* de 10%. Foi utilizada também a métrica *F-Measure* para avaliar os resultados. Esta métrica é a média ponderada entre *precision* e *recall*, sendo seu melhor valor 1 e pior 0. Uma vez que esta métrica é maior que 80% para ambos os classificadores, podemos

concluir que ambos os modelos preditivos prevêem corretamente o tempo de execução de um workflow de acordo com as variáveis de entrada em mais de 80% dos casos. Entretanto, apesar de serem resultados promissores, são necessários mais experimentos com diferentes tipos de workflow para analizar a usabilidade das regras geradas.

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho teve como objetivo investigar diferentes técnicas de tolerância a falhas aplicadas no contexto de workflows científicos executados em nuvens computacionais, além de desenvolver um método que auxilia o cientista na escolha pela TTF que melhor se aplica às tarefas do workflow que será executado. Foi realizado um estudo comparativo destas TTF utilizando-se problemas reais de larga escala modelados como workflows científicos.

Em termos de detecção de falhas, este trabalho propôs o uso de dados fornecidos pelos provedores de serviços de computação em nuvem em conjunto com dados de proveniência coletados pelo SGWfC de forma a, proativamente, evitar a ocorrência de falhas nas execuções das tarefas ou, ao menos, identificar estas falhas o mais cedo possível.

Em termos de mecanismos de correção de falhas, este trabalho implementou as técnicas de Checkpoint-Restart e Replicação de tarefas no SGWfC SciCumulus e as comparou com a técnica básica de Re-execução. Os workflows SciPhy e Montage foram modelados e executados diversas vezes, em variados ambientes computacionais de nuvem e sendo submetidos a diferentes padrões de falhas. Os registros das execuções (13.628) destes 2 workflows foram utilizados para treinar 2 modelos preditivos - árvores de classificação e CN2 - com o objetivo de auxiliar a escolha pela técnica de tolerância a falhas a ser utilizada em cada uma das atividades do workflow. Ambos os modelos foram avaliados e apresentaram métricas de *Precision*, *Recall* e *F-Measure* acima de 0.80, indicando que são efetivos. Entretanto, eles podem ser melhorados no futuro através da execução de novos experimentos, com um conjunto de workflows mais amplo e a coleta de novos registros.

As principais contribuições deste trabalho foram: (i) o desenvolvimento de um modelo de detecção de falhas proativo no SciCumulus, (ii) o desenvolvimento de uma estratégia de Checkpoint-Restart no SciCumulus de baixo custo computacional e que não requer a

prévia alteração dos programas que vierem a ser executados pelo workflow, (iii) o desenvolvimento da técnica de Replicação de tarefas no SciCumulus e (iv) o desenvolvimento de 2 modelos preditivos que podem ser usados por um SGWfC para escolher a técnica de tolerância a falhas mais apropriada para um workflow em específico antes da sua execução.

É importante frisar que alguns dos parâmetros utilizados nos experimentos, como o intervalo entre checkpoints, foram escolhidos de forma puramente *ad-hoc*, ainda que alguns estudos [26, 70] se proponham a encontrar o tamanho do intervalo ótimo entre checkpoints a partir da probabilidade de falhas e tempo de execução esperada para uma determinada tarefa. Tal decisão foi tomada de forma a explicitar que, apesar da escolha ruim para tal parâmetro, que obriga a serem gravados mais *checkpoints* do que ocorreria caso fosse aplicada alguma das abordagens citadas anteriormente, a consequente degradação de desempenho associada a tal escolha não exerce grande impacto na implementação desta estratégia de C-R. Isto ocorre somente devido ao baixo overhead ($< 5\%$) incutido ao processamento da tarefa utilizando a estratégia desenvolvida. Desta forma, foi observado bom desempenho tanto em cenários sem ocorrência de falhas quanto na presença das mesmas.

Como trabalho futuro planeja-se observar o desempenho das técnicas de tolerância a falhas desenvolvidas neste trabalho com novos workflows, em situações reais de falha, além de implementar e analisar o desempenho dos modelos preditivos em tempo de execução. O acoplamento de tal modelo de decisão em um SGWfC pode ser de grande valia uma vez que a escolha de técnicas de tolerância a falha mais adequadas ao contexto em questão tende a trazer melhor tratamento de falhas com menor *overhead*, melhorando a resiliência e diminuindo o tempo de execução e custos associados.

Por fim, espera-se que este trabalho contribua com a ampliação do uso de workflows científicos por parte dos cientistas, auxiliando na obtenção de resultados de forma mais rápida, barata e conveniente, facilitando o uso dos recursos computacionais providos pelas nuvens e permitindo-os focar na condução de seus experimentos.

Referências

[1] Altintas, I.; Berkley, C.; Jaeger, E.; Jones, M.; Ludascher, B.; Mock, S. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on* (2004), IEEE, pp. 423–424.

[2] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I., et al. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[3] Avižienis, A. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, fall joint computer conference* (1967), ACM, pp. 733–743.

[4] Avizienis, A.; Kopetz, H.; Laprie, J.-C. *The Evolution of Fault-Tolerant Computing: In the Honor of William C. Carter*, vol. 1. Springer Science & Business Media, 2012.

[5] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing 1*, 1 (2004), 11–33.

[6] Bala, A.; Chana, I. Autonomic fault tolerant scheduling approach for scientific workflows in cloud computing. *Concurrent Engineering 23*, 1 (2015), 27–39.

[7] Baude, F.; Caromel, D.; Delbé, C.; Henrio, L. A hybrid message logging-cic protocol for constrained checkpointability. In *European Conference on Parallel Processing* (2005), Springer, pp. 644–653.

[8] Berriman, G. B.; Deelman, E.; Good, J. C.; Jacob, J. C.; Katz, D. S.; Kesselman, C.; Laity, A. C.; Prince, T. A.; Singh, G.; Su, M.-H. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *SPIE Astronomical Telescopes+ Instrumentation* (2004), International Society for Optics and Photonics, pp. 221–232.

[9] Boehm, B. W. Verifying and validating software requirements and design specifications. *IEEE software 1*, 1 (1984), 75.

[10] Budhiraja, N.; Marzullo, K.; Schneider, F. B.; Toueg, S. The primary-backup approach. *Distributed systems 2* (1993), 199–216.

[11] Buyya, R.; Yeo, C. S.; Venugopal, S.; Broberg, J.; Brandic, I. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems 25*, 6 (2009), 599–616.

[12] Caltech. Cybershake. In *https://scec.usc.edu/scecpedia/CyberShake_Workflows* (2016). Accessed: 2017-03-01.

[13] Caltech. Montage. In *http://montage.ipac.caltech.edu* (2016). Accessed: 2017-03-01.

[14] Chandy, K. M.; Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3*, 1 (1985), 63–75.

[15] Chillarege, R.; Bhandari, I. S.; Chaar, J. K.; Halliday, M. J.; Moebus, D. S.; Ray, B. K.; Wong, M.-Y. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering 18*, 11 (1992), 943–956.

[16] Clark, P.; Niblett, T. The cn2 induction algorithm. *Machine Learning 3*, 4 (1989), 261–283.

[17] Costa, F.; de Oliveira, D.; Ocaña, K. A.; Ogasawara, E.; Mattoso, M. Enabling re-executions of parallel scientific workflows using runtime provenance data. In *International Provenance and Annotation Workshop* (2012), Springer, pp. 229–232.

[18] Coutinho, R. d. C.; Drummond, L. M.; Frota, Y.; de Oliveira, D. Optimizing virtual machine allocation for parallel scientific workflows in federated clouds. *Future Generation Computer Systems 46* (2015), 51–68.

[19] Cristian, F. Understanding fault-tolerant distributed systems. *Communications of the ACM 34*, 2 (1991), 56–78.

[20] Davidson, S. B.; Freire, J. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1345–1350.

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

[22] Deelman, E. Looking into the future of workflows: The challenges ahead. In *Workflows for e-Science*. Springer, 2007, pp. 475–481.

[23] Deelman, E.; Blythe, J.; Gil, Y.; Kesselman, C.; Mehta, G.; Patil, S.; Su, M.-H.; Vahi, K.; Livny, M. Pegasus: Mapping scientific workflows onto the grid. In *undefined* (2004), Springer, pp. 11–20.

[24] Deelman, E.; Gannon, D.; Shields, M.; Taylor, I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems 25*, 5 (2009), 528–540.

[25] Deelman, E.; Vahi, K.; Juve, G.; Rynge, M.; Callaghan, S.; Maechling, P. J.; Mayani, R.; Chen, W.; da Silva, R. F.; Livny, M., et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems 46* (2015), 17–35.

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

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

[28] Engelmann, C.; Vallee, G. R.; Naughton, T.; Scott, S. L. Proactive fault tolerance using preemptive migration. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on* (2009), IEEE, pp. 252–257.

[29] Fabra, J. Using cloud-based resources to improve availability and reliability in a scientific workflow execution framework.

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

[31] Fu, S.; Xu, C.-Z. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (2007), ACM, p. 41.

[32] Gärtner, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR) 31*, 1 (1999), 1–26.

[33] Graves, R.; Jordan, T. H.; Callaghan, S.; Deelman, E.; Field, E.; Juve, G.; Kesselman, C.; Maechling, P.; Mehta, G.; Milner, K., et al. Cybershake: A physics-based seismic hazard model for southern california. *Pure and Applied Geophysics 168*, 3-4 (2011), 367–381.

[34] Gu, Y.; Wu, C. Q.; Liu, X.; Yu, D. Distributed throughput optimization for large-scale scientific workflows under fault-tolerance constraint. *Journal of grid computing 11*, 3 (2013), 361–379.

[35] Gupta, R.; Beckman, P.; Park, B.-H.; Lusk, E.; Hargrove, P.; Geist, A.; Panda, D. K.; Lumsdaine, A.; Dongarra, J. Cifts: A coordinated infrastructure for fault-tolerant systems. In *Parallel Processing, 2009. ICPP'09. International Conference on* (2009), IEEE, pp. 237–245.

[36] Hiden, H.; Woodman, S.; Watson, P.; Cala, J. Developing cloud applications using the e-science central platform. *Philos Trans A Math Phys Eng Sci. 371*, 1 (2013), 1983.

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

[38] Hoheisel, A. Grid workflow execution service-dynamic and interactive execution and visualization of distributed workflows. In *Proceedings of the Cracow Grid Workshop* (2006), vol. 2, Citeseer, pp. 13–24.

[39] Hu, M.; Luo, J.; Wang, Y.; Veeravalli, B. Adaptive scheduling of task graphs with dynamic resilience. *IEEE Transactions on Computers 66*, 1 (2017), 17–23.

[40] Huang, Y.; Kintala, C.; Kolettis, N.; Fulton, N. D. Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on* (1995), IEEE, pp. 381–390.

[41] Hwang, S.; Kesselman, C. Grid workflow: A flexible failure handling framework for the grid. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on* (2003), IEEE, pp. 126–137.

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

[43] Jain, A.; Ong, S. P.; Chen, W.; Medasani, B.; Qu, X.; Kocher, M.; Brafman, M.; Petretto, G.; Rignanese, G.-M.; Hautier, G., et al. Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience 27*, 17 (2015), 5037–5059.

[44] Kawazoe Aguilera, M.; Chen, W.; Toueg, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Distributed algorithms* (1997), 126–140.

[45] Lamport, L. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, 2 (1977), 125–143.

[46] Laprie, J.-C. Dependability of computer systems: concepts, limits, improvements. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on* (1995), IEEE, pp. 2–11.

[47] Li, Y.; Lan, Z. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on* (2006), vol. 1, IEEE, pp. 8–pp.

[48] Liu, J.; Pacitti, E.; Valduriez, P.; Mattoso, M. A survey of data-intensive scientific workflow management. *Journal of Grid Computing 13*, 4 (2015), 457–493.

[49] Mattoso, M.; Werner, C.; Travassos, G. H.; Braganholo, V.; Ogasawara, E.; Oliveira, D.; Cruz, S.; Martinho, W.; Murta, L. Towards supporting the life cycle of large scale scientific experiments. *International Journal of Business Process Integration and Management 5*, 1 (2010), 79–92.

[50] Melliar-Smith, P. M.; Randell, B. Software reliability: The role of programmed exception handling. In *ACM SIGSOFT Software Engineering Notes* (1977), vol. 2, ACM, pp. 95–100.

[51] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (1989), 541–580.

[52] Nelson, V. P. Fault-tolerant computing: Fundamental concepts. *Computer 23*, 7 (1990), 19–25.

[53] Nguyên, T.; Désidéri, J.-A.; Trifan, L. Applications resilience on clouds. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on* (2012), IEEE, pp. 60–66.

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

[55] Ogasawara, E.; Dias, J.; Silva, V.; Chirigati, F.; Oliveira, D.; Porto, F.; Valduriez, P.; Mattoso, M. Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience 25*, 16 (2013), 2327–2341.

[56] Oinn, T.; Greenwood, M.; Addis, M.; Alpdemir, M. N.; Ferris, J.; Glover, K.; Goble, C.; Goderis, A.; Hull, D.; Marvin, D., et al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18*, 10 (2006), 1067–1100.

[57] Pidd, M. Computer simulation in management science.

[58] Pool, R. The third branch of science debuts. *Science 256*, 5053 (1992), 44.

[59] Quinlan, J. R. Simplifying decision trees. *Int. J. Man-Mach. Stud. 27*, 3 (Sept. 1987), 221–234.

[60] Rappa, M. A. The utility business model and the future of computing services. *IBM systems journal 43*, 1 (2004), 32–42.

[61] Saavedra-Barrera, R.; Culler, D.; Von Eicken, T. Analysis of multithreaded architectures for parallel computing. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures* (1990), ACM, pp. 169–178.

[62] Schneider, F. B. Replication management using the state-machine approach. *Distributed systems 2* (1993), 169–198.

[63] Taylor, I.; Shields, M.; Wang, I.; Harrison, A. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*. Springer, 2007, pp. 320–339.

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

[65] Teylo, L.; de Paula, U.; Frota, Y.; de Oliveira, D.; Drummond, L. M. A hybrid evolutionary algorithm for task scheduling and data assignment of data-intensive scientific workflows on clouds. *Future Generation Computer Systems* (2017).

[66] Vaquero, L. M.; Rodero-Merino, L.; Caceres, J.; Lindner, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review 39*, 1 (2008), 50–55.

[67] von Laszewski, G.; Hategan, M. Java cog kit karajan/gridant workflow guide. Tech. rep., Technical Report, Argonne National Laboratory, Argonne, IL, USA, 2005.

[68] Wozniak, J. M.; Armstrong, T. G.; Wilde, M.; Katz, D. S.; Lusk, E.; Foster, I. T. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on* (2013), IEEE, pp. 95–102.

[69] Wozniak, J. M.; Armstrong, T. G.; Wilde, M.; Katz, D. S.; Lusk, E.; Foster, I. T. Swift/t: Scalable data flow programming for many-task applications. *SIGPLAN Not. 48*, 8 (Feb. 2013), 309–310.

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

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

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