UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
MESTRADO E DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO MARCELO UCHÔA DE ALENCAR

RECONFIGURAÇÃO ELÁSTICA DE COMPONENTES PARALELOS EM NUVENS DE SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO

Fortaleza

2017

JOÃO MARCELO UCHÔA DE ALENCAR

RECONFIGURAÇÃO ELÁSTICA DE COMPONENTES PARALELOS EM NUVENS DE SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO

Tese apresentada à Coordenação do Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará como requisito para a o Doutorado em Ciência da Computação.

Orientador: Prof. Dr. Francisco Heron de Carvalho Junior

Fortaleza

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A353r Alencar, João Marcelo Uchôa de.
Reconfiguração Elástica de Componentes Paralelos em Nuvens de Serviços de Computação de Alto Desempenho / João Marcelo Uchôa de Alencar. – 2017.
134 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2017.
Orientação: Prof. Dr. Francisco Heron de Carvalho Junior.

1. Computação de Alto Desempenho. 2. Desenvolvimento Baseado em Componentes. 3. Nuvens Computacionais. 4. Elasticidade. I. Título.

CDD 005

JOÃO MARCELO UCHÔA DE ALENCAR

RECONFIGURAÇÃO ELÁSTICA DE COMPONENTES PARALELOS EM NUVENS DE SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO

João Marcelo Uchôa de Alencar

Tese submetida à Coordenação do Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará em 31/08/2017 como requisito para a obtenção do grau de Doutor em Mestrado e Doutorado em Ciência da Computação, na cidade de Fortaleza, pela banca examinadora assim constituída:

Prof. Dr. Francisco Heron de Carvalho Junior
Orientador

Prof. Dr. Antônio Tadeu Azevedo Gomes

Profa. Dra. Rossana Maria de Castro Andrade

Prof. Dr. Lincoln Souza Rocha

Prof. Dr. Paulo Herinque Mendes Maia

Fortaleza

2017

Dedico este trabalho a Maria do Carmo Alencar...

Agradecimentos

Inicio os agradecimentos lembrando o esforço dos meus pais, João e Josefa, durante toda a minha criação, sempre permitindo que fosse em busca dos meus sonhos. Da mesma forma que meus irmãos, sempre recebi o apoio necessário e o exemplo presente de que o trabalho duro e honesto rende recompensas legítimas.

Sobre minha esposa, Germana, sua tenacidade e compreensão também estão presentes nos resultados deste trabalho. Nem sempre fui capaz de ser o marido perfeito, devido às diversas obrigações que vida profissional que escolhi impõe, mas sua paciência e amor incondicionais foram imprescindíveis para a conclusão desta Tese.

O campus de Quixadá, minha segunda casa, repleto de colegas prontos para ajudarem, seja através de conselhos ou até mesmo colocando a mão na massa. Sem o apoio do corpo docente e a dedicação do corpo discente e dos servidores, meu caminho teria sido muito mais árduo. Com o título de doutor, espero poder contruibuir muito mais para o desenvolvimento do campus.

Você se torna professor sendo aluno. E fui aluno de ótimos professores. A memória e o espaço não permitem listar todos, mas gostaria de agradecer aos professores da UECE, onde fiz a graduação, e aos docentes do MDCC-UFC, sede do meu mestrado e agora doutorado. Sua dedição foi a mais forte inspiração para a carreira que escolhi seguir. Em especial, agradeço a professora Rossana, que foi a primeira a me acolher e apoiar na UFC.

Sem meus amigos e companheiros do LAB03, mais frequentes teriam sido os momentos de desespero. O conhecimento se constrói de forma coletiva, sendo que na nossa comunidade a união é a principal característica. Tenho certeza que o futuro nos reserva maiores surpresas e desse grupo sairão grandes profissionais.

Finalizando, toda Tese é um trabalho de no mínimo quatro mãos. Reconheço o esforço e dedicação do professor Francisco Heron, um exemplo de honestidade e conhecimento acadêmico cuja convivência terá impacto constante no decorrer da minha vida profissional e acadêmica. Compartilho com ele a responsabilidade do trabalho aqui apresentado, sendo que expresso meus sinceros agradecimentos.

The more you know the less you believe,
The more you have the more it takes today.
- Paul Hewson

Resumo

Os pesquisadores que desejam executar aplicações científicas possuem uma vasta opção de infraestruturas computacionais como supercomputadores e nuvens. Essas infraestruturas são de natureza compartilhada, é norma existirem mudanças no estado dos recursos durante a execução de um programa paralelo. É importante que os usuários finais e os desenvolvedores tenham meios de adaptar e controlar a execução para garantir o cumprimento de requisitos de qualidade de serviço. No caso específico das nuvens, a reconfiguração é habilitada pelo conceito de elasticidade. Nesta Tese, apresentamos uma arquitetura para um arcabouço que permita aos atores envolvidos definir políticas e mecanismos de controle em tempo de execução para a elasticidade de componentes de sistemas paralelos, alterando o conjunto de recursos alocados. Utilizamos como ambiente de execução a nuvem de componentes HPC Shelf. Esse ambiente apresenta um sistema de contratos contextuais que permite a descrição e alocação adequada de componentes e plataformas virtuais, definindo a combinação otimizada de acordo com as características do programas paralelos e os recursos disponíveis nas plataformas. Os contratos contextuais também permitem a definição de contratos de qualidade de serviço de acordo com prioridades e requisitos dos pesquisadores. Dessa forma, o ambiente resultante da HPC Shelf com controle de reconfiguração elástica amplia as opções dos atores envolvidos, garantindo que o contrato de qualidade de serviços dos componentes seja respeitado diante de informações incompletas e de mudanças no ambiente.

Palavras-chave: Computação de Alto Desempenho. Componentes. Elasticidade. Nuvens Computacionais.

Abstract

Researchers willing to run scientific applications have a vast choice of computing infrastructures such as supercomputers and clouds. Since these infrastructures are of shared nature, it is usual to face variability in the state of resources during the execution of a parallel program. End users and developers should have the capability to adapt and control execution to ensure compliance with quality of service requirements. In the particular case of clouds, the reconfiguration is enabled by the concept of elasticity. In this Thesis, we present an architecture for a framework that allows the stakeholders to define policies and mechanisms of control at runtime for the elasticity of components of parallel systems, changing the set of allocated resources. We use the HPC Shelf cloud of components as the execution environment. This cyberinfrastructure presents a contextual contract system that allows the proper description and allocation of components and virtual platforms, defining the optimized match according to the characteristics of the parallel programs and the resources available on the platforms. Contextual contracts also allow the definition of quality of service contracts according to researchers' priorities and requirements. The resulting HPC Shelf environment with elastic reconfiguration control expands the options for the stakeholders, ensuring that the runtime respects the component service quality contract in the face of incomplete information and fluctuations in the environment.

Keywords: High Performance Computing. Components. Elasticity. Cloud Computing.

Lista de Figuras

2.1 Arquitetura interna de um nó. 25
2.2 Topologia do Cluster do CENAPAD-UFC. 27
2.3 Distribuição de filas em um gerenciador de recursos. 29
2.4 Esquema geral de uma grade computacional. 30
2.5 Classificação das nuvens. 32

3.1 Padrão de projeto *uses/provides*. 47
3.2 Diagrama do padrão de projeto SCMD. 50
3.3 Visão Geral de uma Infraestrutura com Suporte a CQoS. 51
3.4 Composição de Componentes no Fractal. 53
3.5 Interação entre gerentes e controladores. 55
3.6 Componente Autonômico Passivo. 57
3.7 Componente Autonômico Ativo. 58

4.1 O laço MAPE. 62
4.2 Mapeamento de Aplicações CAD em várias plataformas. 67
4.3 Submissão de aplicação no CodeCloud. 68
4.4 Unidade de execução paralela de acordo. 70
4.5 API e *Middleware*. 72
4.6 Arquitetura da AutoElastic. 73

5.1 Exemplo de composição por sobreposição. 78
5.2 Conector na nuvem HPC Self. 81
5.3 Arquitetura da HPC Shelf. 87
5.4 Registro de Componentes no Core. 88
5.5 Resolução de Componentes pelo Core. 89

6.1 Reconfiguração em código maleável. 97
6.2 Reconfiguração em código evolutivo. 97
6.3 Ligação de componentes em um sistema computacional. 99

6.4 Ligação de componentes em um sistema computacional maleável. 102

6.5 Ligação de componentes em um sistema computacional evolutivo. 107

6.6 Estrutura interna de um componente Hash elástico. 108

7.1 Tempo de Execução sem Reconfiguração. 114

7.2 Eficiência sem Reconfiguração. 115

7.3 Tempo de Execução de Componente de Sistema Maleável com Prioridade de Tempo. 116

7.4 Eficiência de Componente de Sistema Maleável com Prioridade de Tempo. 117

7.5 Custo de Componente de Sistema Maleável com Prioridade de Tempo. . . 118

7.6 Tempo de Execução do Componente Evolutivo. 120

7.7 Custo de Componente Evolutivo. 121

Lista de Tabelas

2.1 Classificação de tarefas paralelas de acordo com quantidade nós de processamento utilizada. 38

4.1 Visão geral da elasticidade em CAD a nível do desenvolvedor. 74

6.1 Chamadas da biblioteca de elasticidade para aplicações evolutivas. 98

6.2 Funções para especialização do laço MAPE. 106

7.1 Comportamento do Sistema Computacional sem Reconfiguração. 112

7.2 Impacto da Variação dos Pesos no Cenário da Carga Variável. 119

Lista de Abreviaturas e Siglas

CAD Computação de Alto Desempenho

CBSE *Component-Based Sofware Engineering*

CCA *Common Component Architecture*

CQoS *Computational Quality of Service*

FLOPs *Floating Points Operations per Seconds*

FPGA *Field-Programmable Gate Array*

GCM *Grid Component Model*

GPU *Graphic Processing Units*

HPE *Hash Programming Environment*

HTS *Hash Type System*

IaaS *Infrastructure as a Service*

MAPE Monitorar Analisar Planejar e Executar

MIC *Many Integrated Cores*

MPI *Message Passing Interface*

MPMD *Multiple Program Multiple Data*

MPP *Massive Parallel Processors*

NIC *Network Interface Card*

PaaS *Platform as a Service*

QoS *Quality-of-Service*

SaaS *Software as a Service*

SCMD *Single Component Multiple Data*

SIMD *Single Instruction, Multiple Data*

SLURM *Simple Linux Utility for Resource Management*

SPMD *Single Program Multiple Data*

Sumário

Lista de Figuras **10**

Lista de Tabelas **11**

Lista de Abreviaturas e Siglas **12**

1 Introdução **17**

1.1 Motivação da Pesquisa 20

1.2 Objetivos 21

1.2.1 Objetivo Geral 22

1.2.2 Objetivo Específicos 22

1.3 Metodologia 22

1.4 Organização da Tese 23

2 Computação de Alto Desempenho: Fundamentos e Tecnologia **24**

2.1 Plataformas de Computação de Alto Desempenho 24

2.1.1 Visão Geral de um Supercomputador Moderno 24

2.1.2 Grades Computacionais 29

2.1.3 Nuvens Computacionais 31

2.1.4 Nuvens versus Grades 34

2.2 Modelos de Programação 35

2.2.1 Modelo de Variáveis Compartilhadas 35

2.2.2 Modelo de Troca de Mensagens 36

2.3 Tipos de Tarefas Paralelas 37

2.3.1 Objetivos do Escalonamento 37

2.3.2 Classificação 38

2.4 Escalabilidade de Programas Paralelos 39

2.4.1 Métricas de Desempenho 40

2.4.2 Escalabilidade de Sistemas Paralelos 41

2.5 Conclusões 43

3 Computação de Alto Desempenho Baseada em Componentes (CBHPC) 45
3.1 CCA 46
3.1.1 O Padrão de Projeto *Uses/Provides* 46
3.1.2 O Padrão de Projeto SCMD 49
3.1.3 Reconfiguração Dinâmica de Componentes 50
3.2 GCM 52
3.2.1 Composição Hierárquica 53
3.2.2 Comunicação Coletiva 53
3.2.3 Gerência Autonômica e Adaptação no GCM 54
3.2.3.1 Esqueletos Comportamentais 55
3.2.3.2 Arquitetura de Componentes Autonômicos 56
3.3 Conclusões 57

4 Elasticidade em Nuvens Computacionais 59
4.1 Elasticidade para Aplicações Comerciais 60
4.1.1 Conceitos Gerais 60
4.1.2 Aplicações Comerciais Elásticas 60
4.1.3 Gerência de Elasticidade 61
4.1.3.1 O Laço MAPE 61
4.1.3.2 Técnicas de Elasticidade 62
4.2 Elasticidade em Computação de Alto Desempenho 64
4.2.1 Definição de Elasticidade para CAD 64
4.2.2 Gerência Elástica de Recursos 66
4.2.3 Suporte à Elasticidade em Nível de Programação 67
4.2.3.1 CodeCloud 67
4.2.3.2 ParMod 69
4.2.3.3 Elastic MPI 70
4.2.3.4 Malleable MPI 71
4.2.3.5 Cloudline 71
4.2.3.6 AutoElastic 72
4.2.3.7 Análise Comparativa 73
4.3 Conclusões 74

5 A Nuvem de Componentes HPC Shelf **75**

5.1 Sistemas de Computação Paralela 75

5.2 O Modelo de Componentes Hash 77

5.2.1 Componentes Paralelos e Composição por Sobreposição 77

5.2.2 HPE 78

5.3 As Espécies de Componentes da HPC Shelf 80

5.4 Atores Envolvidos 82

5.5 O Sistema de Contratos Contextuais 83

5.6 Arquitetura da HPC Shelf 86

5.7 Qualidade de Serviço em Contratos Contextuais 90

5.8 Conclusões 93

6 Um Arcabouço para Construção de Aplicações de Computação de Alto Desempenho Elásticas **94**

6.1 Motivação 95

6.2 Reconfiguração Elástica 96

6.3 Arquitetura do Ambiente 98

6.3.1 Reconfiguração Elástica de Componentes na HPC Shelf 101

6.3.1.1 Sistemas Maleáveis 101

6.3.1.2 Sistemas Evolutivos 106

6.3.2 Estrutura Interna de Componentes Paralelos Elásticos 107

6.4 Conclusões 109

7 Estudos de Casos **110**

7.1 Ambiente de Testes 110

7.2 Sistemas Computacionais Maleáveis 111

7.2.1 Comportamento do Sistema sem Reconfiguração 111

7.2.2 Comportamento do Sistema com Reconfiguração 114

7.3 Sistemas Computacionais Evolutivos 119

7.4 Conclusões 120

8 Conclusão **122**

8.1 Contribuições 123

8.2 Desafios 123

8.2.1 Reputação de Componentes 123
8.2.2 Avaliação de Padrões de Paralelismo e Plataformas 124

Referências Bibliográficas **126**

1. Introdução

Computação de Alto Desempenho (CAD) se refere à prática de consolidação de poder computacional com o objetivo de alcançar desempenho maior do que computadores típicos, de uso pessoal ou servidores, na resolução de problemas computacionalmente intensivos de interesse das ciências, engenharias e ramos de negócios (GUPTA, 2014). O impacto das tecnologias de CAD em diversas áreas é notório, especialmente a partir dos anos 1970 com o surgimento dos primeiros supercomputadores propriamente ditos, para viabilizar aplicações científicas estratégicas e na indústria militar, instalados em alguns poucos laboratórios financiados pelos governos de alguns países economicamente e tecnologicamente hegemônicos, bem como considerados tecnologia estratégica. Nos anos 1990, com o surgimento da computação paralela em *clusters*, utilizando *hardware* de prateleira, observamos a disseminação do uso desse tipo de tecnologia em vários centros de pesquisa, universidades, indústrias e sistemas corporativos, em diversos países, motivando a implementação de uma grande variedade de aplicações. Dessa forma, as tecnologias de CAD tornaram-se responsáveis por grande parte dos avanços em campos de conhecimento tais como cosmologia, dinâmica molecular, química quântica, previsões climáticas (e.g. aquecimento global), genética humana, descoberta de medicamentos, dentre outras. Na área de negócios, seu impacto é mais recente, especialmente no mercado financeiro (econofísica) e em análise de dados em sistemas comerciais. O interesse estatal também é muito forte, especialmente para os propósitos de sistemas de segurança, vigilância e na indústria militar (e.g. testes *in silico* de ogivas nucleares, utilizando métodos de simulação).

A princípio, a natureza da Computação de Alto Desempenho consiste em lidar com problemas em *larga escala*, cenário no qual pesquisadores se esforçam para aprimorar a precisão da solução para um dado modelo físico fixo. Essa abordagem pode ser obtida com o avanço tecnológico do *hardware* utilizado, o que permite a utilização de processadores com frequências maiores e com vários núcleos embutidos, além de redes de comunicação com menor latência e maior largura de banda. Entretanto, para atender aos interesses da ciência moderna, de caráter fortemente interdisciplinar, a Computação de Alto Desempenho precisa considerar problemas de *amplo escopo* (GIL, 2008). Por amplo escopo, entendemos problemas que além de exigir precisão da solução requisitam a reformulação do modelo físico adotado, incluindo cenários e entidades não previstos no primeiro momento. Essa reformulação não depende apenas do aprimoramento do *hardware*, o código da aplicação também precisa evoluir, apresentando maior complexidade e heterogeneidade, porém sem perder a manutenibilidade.

Visando atender as demandas das abordagens utilizadas pelos pesquisadores, o ecossistema de CAD evoluiu no decorrer do tempo. As primeiras infraestruturas utilizadas foram os *supercomputadores*, que a partir da década de 1970 adotaram o paradigma SIMD (*Single Instruction, Multiple Data* (KUMAR, 1994)) nas arquiteturas de suas unidades centrais de processamento. Esse tipo de processador, classificado como vetorial, possui várias unidades de execução que realizam uma mesma instrução em um vetor de uma dimensão armazenado na memória principal. Já no início década de 1980, os computadores vetoriais passaram a ter mais de um processador acessando uma memória global compartilhada (multiprocessadores vetoriais). Os MPPs (*Massive Parallel Processors*) representam o auge do desenvolvimento da arquitetura dos supercomputadores clássicos, unindo, através de um barramento de alto desempenho, vários processadores vetoriais, cada um com um banco de memória particular. Para os supercomputadores, cada fabricante desenvolvia a pilha completa de *hardware* (processadores, memória, rede de conexão, etc) e *software* (sistema operacional, linguagens, compiladores,etc). Os programas eram dependentes do computador para o qual tinham sido desenvolvidos, ou seja, a portabilidade era limitada.

O alto custo e a pouca portabilidade dos supercomputadores incentivaram a criação de máquinas para CAD que utilizavam processadores, memórias e outras peças não muito diferentes daquelas encontradas nos computadores pessoais, em geral mais acessíveis e com padrões bem definidos. Na década de 1990, o *Cluster Beowulf* (STERLING, 2002) se consolidou como um sistema que unia vários computadores pessoais com o objetivo de oferecer desempenho próximo a um supercomputador proprietário mas com apenas uma fração do custo. Cada PC tinha apenas um processador e memória local. A interação entre processos em computadores diferentes era feita através da troca de mensagens em uma rede de conexão local, sendo assim uma hierarquia de memória distribuída. Desta forma, clusters se enquadram na classificação MIMD (*Multiple Instruction, Multiple Data streams*).

Em paralelo ao desenvolvimento do *hardware* para CAD, no final da década de 1990 surgiram discussões sobre maneiras de fornecer acesso a essas máquinas para um público maior de pesquisadores e como otimizar o seu uso, em especial diminuindo os períodos de ociosidade. Supercomputadores e *clusters* eram adquiridos e administrados por grupos de pesquisa ou instituições independentes. Como é natural no desenvolvimento científico, havia períodos de intensa utilização, intercalados por intervalos de análise dos resultados das simulações. Durante essa análise, a máquina ficava a maior parte do tempo ocioso. Esses ciclos inutilizados poderiam muito bem ser aproveitados por outros pesquisadores. O que os cientistas também perceberam é que unindo seus recursos computacionais, problemas que antes exigiriam muito tempo para serem atacados individualmente seriam resolvidos de maneira mais rápida através da colaboração de diversos supercomputadores

espalhados geograficamente. Da sinergia apresentada nesse cenário, surgiu o conceito de *grades computacionais* (FOSTER; KESSELMAN, 2003), que também tinham como meta fomentar a colaboração científica.

Mesmo com o advento das grades e do compartilhamento de recursos, acesso a supercomputadores e *clusters* era restrito no início da década de 2000. Somente pesquisadores pertencentes a instituições de pesquisa consolidadas, com orçamento suficiente para aquisição e manutenção de recursos para CAD, tinham acesso direto à supercomputadores e *clusters*. Em situações nas quais o acesso remoto era permitido através das grades computacionais, a alocação de recursos para um projeto de pesquisa envolvia a análise por parte de comitês científicos. As *nuvens computacionais* (ZHANG; CHENG; BOUTABA, 2010), inicialmente oriundas da computação comercial, surgiram como opção para pesquisadores com requisitos de CAD emergentes. Adotando o paradigma de infraestrutura como serviço, usuários interessados em CAD possuem uma alternativa de baixo custo para estabelecimento de *clusters* de máquinas virtuais ao mesmo tempo que obtêm maior flexibilidade na configuração da pilha de *hardware* e *software*. A maior parte dessa flexibilidade advém da *elasticidade* permitida pelas nuvens. Elasticidade em nuvens é a alocação e liberação dinâmica de recursos sem a intervenção direta do usuário final. A abstração resultante é a ilusão de um conjunto de recursos infinito que podem ser adquiridos e liberados a qualquer momento durante a execução (MELL; GRANCE, 2011).

A elasticidade fornecida pelas nuvens é um característica importante para aplicações reconfiguráveis (GALANTE; BONA, 2012). Por reconfiguração elástica, entendemos a capacidade de uma aplicação CAD alterar o conjunto de recursos alocados durante a execução. Código reconfigurável pode requisitar mais recursos para uma nuvem elástica sem a intervenção do usuário ou administrador. A reconfiguração não trata apenas o aumento do número de recursos para melhoria de desempenho, mas também o caso inverso, no qual uma aplicação libera recursos quando não são mais necessários visando o controle do custo de execução. Por ter impacto no desempenho e no custo da execução, o mecanismo de controle da reconfiguração elástica é responsável pela percepção final que o pesquisador tem da execução de sua aplicação.

De acordo com a evolução descrita, na atualidade um pesquisador que faz uso das técnicas de ciência computacional através da execução de aplicações de Computação de Alto Desempenho tem acesso a várias infraestruturas distintas. O cenário se expande ainda mais visto que as estações de trabalho modernas oferecem poder computacional equivalente ao permitido por máquinas específicas para CAD da década de 1990. Os supercomputadores ou *clusters* estão disponíveis nas universidades ou centros de pesquisa, e apesar de serem plataformas de uso compartilhado, apresentam quantidade de FLOPS (*Floating Points Operations per Seconds*) suficiente para simular sistemas de alta complexidade provenientes das várias áreas do conhecimento humano. O advento das nuvens

computacionais nos últimos anos fornece ainda mais opções com flexibilidade de custo e alto poder de processamento. Essa heterogeneidade é importante para a diminuição dos custos, entretanto acarreta o desafio de desenvolver aplicações a partir dos artefatos de *software* otimizados para cada plataforma, ao mesmo tempo que se busca garantir que a execução seja guiada pelos requisitos dos pesquisadores.

1.1 Motivação da Pesquisa

A diversidade de plataformas, apesar de benéfica ao permitir a resolução de problemas maiores, também acarreta dificuldades na gerência dos recursos. O usuário precisa decidir qual infraestrutura utilizar para executar suas aplicações e essa decisão não é simples. Podem existir várias versões para uma mesma aplicação, cada uma adequada para um tipo específico de plataforma. Portanto, é necessário utilizar o conhecimento do desenvolvedor da aplicação (algoritmos e estruturas de dados), do mantenedor da infraestrutura (configuração do *hardware*) e do usuário final (características da entrada) para tomar uma decisão adequada (HALL; GIL; LUCAS, 2008).

Uma das maneiras para lidar com o problema da diversidade é adotar um paradigma de desenvolvimento que permita a representação uniforme dos diferentes elementos do sistema. O ambiente HPE (*Hash Programming Environment*) (CARVALHO JUNIOR, 2007) permite representar os elementos de um ambiente de Computação de Alto Desempenho através de componentes. No HPE, tanto o código a ser executado quanto a plataforma de execução são representados como componentes. Esse ambiente possui um serviço de catálogo de componentes (o Core) que armazena informações sobre os componentes disponíveis e executa um algoritmo de resolução capaz de formar a composição adequada para uma aplicação de Computação de Alto Desempenho. Originalmente desenvolvido para *clusters* e supercomputadores, na atualidade o HPE está evoluindo para a HPC Shelf, uma nuvem de componentes para Computação de Alto Desempenho.

Escolher a melhor combinação de código e plataforma é o primeiro passo para aprimorar a utilização de infraestruturas de Computação de Alto Desempenho. Com as descrições dos artefatos de *hardware* e *software*, além dos requisitos e prioridades dos pesquisadores, é possível definir um *contrato de qualidade de serviço* (QoS - *quality of service*) que representa uma previsão do comportamento de parâmetros como desempenho e custo financeiro (ROSA RIGHI, 2017). Esse contrato pode ser gerado de maneira automática de acordo com descrições dos componentes plataforma e computação, sendo que a prática demonstra que a combinação de técnicas de análise e conhecimento empírico do desenvolvedor permitem estabelecer valores adequados para a previsão do comportamento da execução de determinada computação em uma plataforma específica (SHUDLER, 2017).

Seja qual for a técnica para definição de contratos, existem situações que colocam em risco o seu cumprimento. Primeiro, apesar do esforço legítimo dos atores que fornecem a descrição dos artefatos, informações imprecisas na forma de valores de entrada mal definidos podem levar a contratos mal formados. Independente da viabilidade do contrato, o ambiente de execução deve atuar para priorizar os requisitos dos pesquisadores. Segundo, não podemos considerar que o estado das infraestruturas é estático durante a execução de uma aplicação (FAN, 2012). Em especial no caso das nuvens, como são recursos compartilhados através de virtualização, o aspecto multi-inquilino possibilita que outras aplicações utilizando a infraestrutura física ao mesmo tempo possam interferir na execução e colocar em risco o contrato. Essa interferência pode ocorrer de várias formas visto que apesar de ser possível alocar núcleos de processamento e memória de maneira exclusiva para cada processo, as arquiteturas atuais não permitem particionar outros elementos da plataforma. Por exemplo, a maioria das arquiteturas atuais fornecem uma quantidade limitada de memória cache para cada núcleo, porém a maior parte dessa memória é compartilhada entre todo o processador. Um processo em execução em um núcleo pode invalidar uma linha da *cache* compartilhada que ainda está em uso por outro processo em um núcleo distinto, causando atrasos na execução. De maneira semelhante, processos de aplicações CAD podem concorrer pelo acesso ao barramento de memória ou à rede interconexão do *cluster*. Em busca de garantir os requisitos de QoS, é preciso adaptar a execução a mudanças no estado da plataforma.

Considerando o cenário atual da CAD e a importância dos contratos de QoS para satisfação das expectativas dos pesquisadores, assumimos nesta Tese o problema de modelar um arcabouço para definição de mecanismos de reconfiguração elástica que visem adaptar a execução de componentes paralelos em direção ao cumprimento dos requisitos presentes no contrato de QoS do sistema formado pelo código de Computação de Alto Desempenho e a plataforma paralela na qual executa. Para resolver tal problema, delimitamos objetivos que devem ser alcançados para o estabelecimento de um arcabouço com as caraterísticas necessárias.

1.2 Objetivos

Dado o contexto e motivação apresentados nas seções anteriores, precisamos considerar a viabilidade de um ambiente de execução para Computação de Alto Desempenho com suporte à reconfiguração elástica para manutenção de contratos de QoS. Nossa hipótese é que tal sistema é viável, sendo que utilizando tecnologias e paradigmas da engenharia de *software* baseada em componentes, podemos atingir os objetivos a seguir.

1.2.1 Objetivo Geral

O objetivo desta Tese é fornecer um arcabouço (*framework*) orientado a componentes que permita aos atores de um ambiente de Computação de Alto Desempenho controlar a qualidade de serviço da execução através da reconfiguração elástica de componentes que representam o software e o hardware paralelo.

1.2.2 Objetivo Específicos

Além do objetivo geral da Tese, os seguintes objetivos específicos são definidos:

- Apresentar um panorama atual das tecnologias de Computação de Alto Desempenho, descrevendo os detalhes arquiteturais de uma plataforma contemporânea e os modelos de programação empregados.
- Levantar o estado da arte em elasticidade para aplicações paralelas, consolidando as soluções existentes de acordo com suas qualidades e limitações.
- Discutir os conceitos fundamentais da nuvem de componentes HPC Shelf, apresentando os mecanismos já existentes para seleção e classificação de componentes de acordo com a qualidade de serviço e propor extensões necessárias para suportar a reconfiguração elástica.
- Propor uma arquitetura para reconfiguração elástica de componentes de computações de alto desempenho.
- Permitir a definição e o reuso de artefatos de *software* que habilitem o controle da elasticidade em aplicações paralelas.

1.3 Metodologia

O primeiro passo para o trabalho apresentado nesta Tese foi uma revisão bibliográfica sobre ambientes de execução de aplicações paralelas na nuvem com suporte à elasticidade. O objetivo foi levantar o estado da arte e encontrar possíveis limitações nas soluções existentes. A partir desse estudo e da definição da contribuição, propusemos o uso de um ambiente concebido previamente que pudesse servir como base para os aprimoramentos propostos. A HPC Shelf apresentou-se como uma opção adequada, visto que essa solução já previa os serviços básicos para alocação de recursos e execução de aplicações, além de adotar as boas práticas de engenharia de *software* através do desenvolvimento orientado

a componentes. A partir da definição das tecnologias de suporte, validamos o *framework* proposto analisando a execução de aplicações no ambiente, verificando se as ações de reconfiguração são executadas de maneira correta e se o resultado final é vantajoso para os usuários e desenvolvedores em termos de cumprimento do contrato de qualidade de serviço.

1.4 Organização da Tese

O restante desta Tese está organizado da seguinte forma. No Capítulo 2, apresentamos a fundamentação teórica sobre Computação de Alto Desempenho. No Capítulo 3, descrevemos abordagens já existentes para Computação de Alto Desempenho baseada em componentes, incluindo uma exposição de como essas soluções adotam mecanismos dinâmicos de reconfiguração. O Capítulo 4 aprofunda a definição do conceito de elasticidade para as nuvens e como o estado da arte é aplicado para Computação de Alto Desempenho. Já que utilizamos a nuvem de componentes HPC Shelf como base para esta Tese, seus fundamentos são apresentados no Capítulo 5. O Capítulo 6 apresenta o *framework* proposto, enquanto o Capítulo 7 oferece estudos de caso para comprovar a viabilidade do modelo. As conclusões, com as contribuições e limitações da Tese, além de trabalhos futuros, estão no Capítulo 8.

2. Computação de Alto Desempenho: Fundamentos e Tecnologia

Neste capítulo, apresentamos a fundamentação teórica para a contribuição apresentada nesta Tese. Começamos com uma discussão sobre as infraestruturas computacionais utilizadas para executar aplicações científicas (Seção 2.1). Em seguida, apresentamos os principais modelos de programação para Computação de Alto Desempenho (Seção 2.2) e como os programas paralelos podem ser classificados em relação a adaptação em tempo de execução (Seção 2.3). Uma modelagem analítica de programas paralelos permite contextualizar os limites da elasticidade para aplicações de alto desempenho (Seção 2.4). Discutimos as relações entre os assuntos do capítulo na Seção 2.5.

2.1 Plataformas de Computação de Alto Desempenho

As plataformas de computação paralela evoluíram de forma radical no decorrer dos anos. A partir dos supercomputadores vetoriais da década de 1970 (RUSSELL, 1978), passando pelos MPPs (*Massive Parallel Processors*) na década de 1980 até o *Cluster Beowulf* (STERLING, 2002) nos anos 1990, temos na atualidade uma arquitetura híbrida baseada em *clusters* com nós de processamento com processadores de vários núcleos e associados a dispositivos aceleradores, dentre os quais merecem destaque as GPUs (*Graphic Processing Units*) (FAN, 2004) , MICs (*Many Integrated Cores*) (DURAN; KLEMM, 2012) e FPGAs (*Field-Programmable Gate Array*) (HERBORDT, 2007) .

Sendo o foco deste trabalho aplicações científicas, apresentamos nessa seção uma visão geral de um supercomputador moderno. Além da descrição de uma máquina que trabalha de forma independente, as duas plataformas distribuídas utilizadas em Computação de Alto Desempenho (grades e nuvens) também são estudadas.

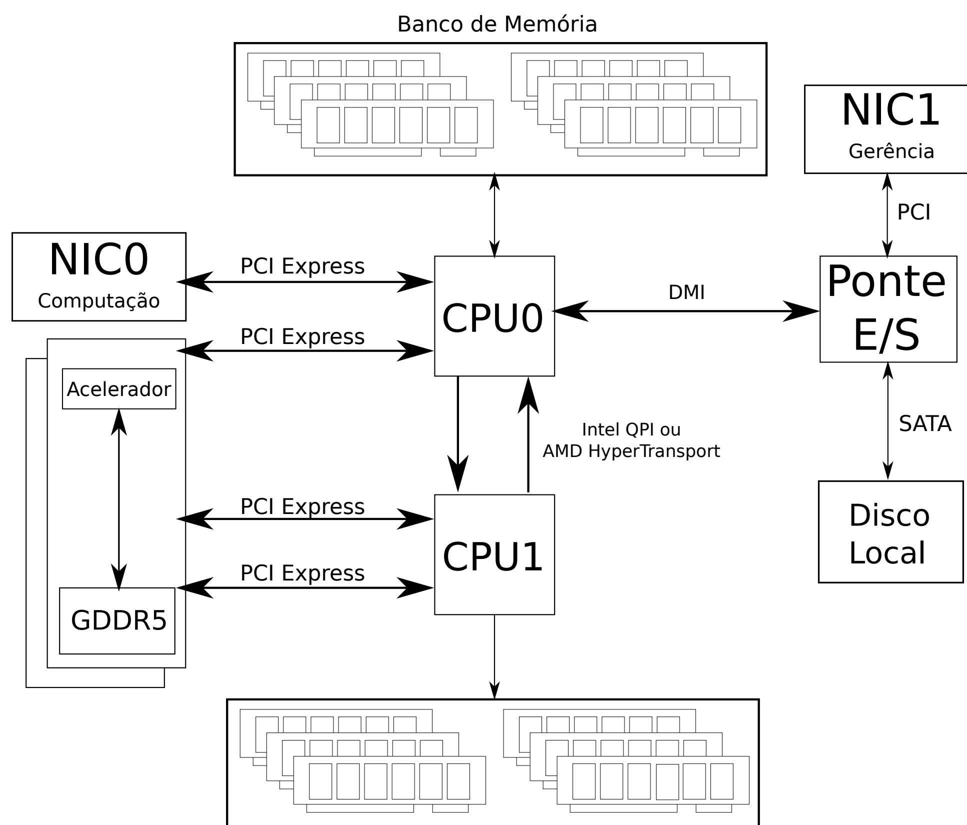
2.1.1 Visão Geral de um Supercomputador Moderno

Um supercomputador moderno é uma evolução dos primeiros *clusters computacionais*. Em outras palavras, arquiteturas baseadas em *cluster* são predominantes, mas alguns elementos dos supercomputadores clássicos com processamento vetorial ainda persistem. Apresentamos aqui as principais características de um supercomputador moderno.

Desenvolvemos nossa descrição a partir do estudo dos sistemas descritos na lista TOP500 (`www.top500.org`) e da máquina instalada no CENAPAD-UFC (`www.cenapad.ufc.br`).

A menor parte de um supercomputador moderno que executa uma imagem do sistema operacional é chamada de *nó de processamento*. A Figura 2.1 é uma representação em alto nível da arquitetura interna. Nessa figura, podemos observar dois processadores (CPU0 e CPU1). Essa quantidade de *chips* pode variar. Cada processador tem acesso ao banco de memória, que apesar de aparecer como duas entidades separadas na imagem, trata-se de um espaço de endereçamento global. Além disso, dentro de cada processador existem vários núcleos (*cores*) que compartilham memória cache (oculta na ilustração para efeito de simplificação). Os processadores podem se comunicar entre si através de um protocolo proprietário.

Figura 2.1: Arquitetura interna de um nó.



Fonte: Adaptado de (DONGARRA, 2013).

Os processadores se comunicam com outros periféricos através de barramentos *PCI Express* (alto desempenho), PCI apenas ou SATA. No caso dos dois últimos barramentos, a comunicação é feita por intermédio de um *chip* especializado para controlar E/S (Ponte E/S). A comunicação entre a ponte e as CPUs é feita por acesso direto à memória, sendo esta um recurso que pode ser fonte de contenção. Os aceleradores como GPUs e MICs são componentes dedicados capazes de executar certos tipos de cálculos com

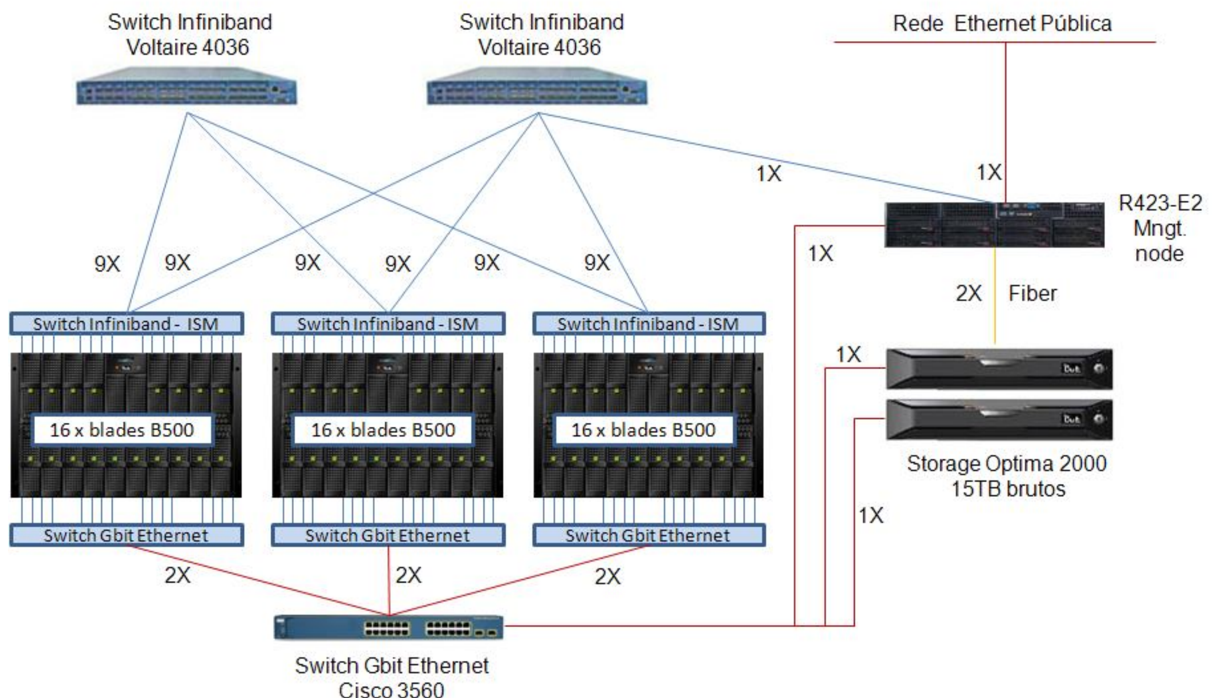
melhor desempenho do que CPUs tradicionais. Existe toda uma área de estudo dedicada aos diferentes tipos de aceleradores computacionais (GELADO, 2010), mas não são o foco deste trabalho.

Para otimizar a execução de aplicações paralelas que fazem uso de troca de mensagens, a configuração de rede em um *cluster* para Computação de Alto Desempenho pode apresentar duas redes (interfaces NIC0 e NIC1) , com tecnologias diferentes. A rede de gerência utiliza tecnologia Ethernet e é usada para *login* remoto e sistemas de arquivos distribuídos. Ela não tem nenhuma diferença em relação a redes locais tradicionais, exceto que costuma utilizar a última versão do padrão Ethernet (atualmente, 100-Gigabit_Ethernet) e utiliza comutadores (*switches*) de qualidade superior. O padrão Ethernet só determina os protocolos de acesso ao enlace. Os outros protocolos necessários para comunicação (IP para rede e TCP/UDP para transporte) são implementados por *software* no *kernel* do sistema operacional. A camada de *software* entre a aplicação e o enlace não é motivo de preocupação para as aplicações tradicionais. Porém, para aplicações de Computação de Alto Desempenho, o *software* implica uma latência que é prejudicial ao desempenho dessas aplicações. Em busca de diminuir a latência, a rede de computação utiliza protocolos que minimizam a presença de *software* na pilha, tentando implementar o máximo possível em *hardware*. Um exemplo é a tecnologia Infiniband (CHEN, 2011), na qual até a camada de transporte a comunicação é tratada diretamente pelo adaptador. Dessa forma, mais do que a largura de banda, a latência é otimizada, requisito crucial para o desempenho de aplicações de Computação de Alto Desempenho. A desvantagem é que o adaptador se torna mais complexo, aumentando o custo de implantação de redes Infiniband.

A Figura 2.2 apresenta a organização de rede do *cluster* do CENAPAD-UFC. Nesse projeto, os nós de processamentos são construídos em *blades*, que são agrupados em *chassis*, gabinetes com comutadores Ethernet e Infiniband embutidos. Além dos nós que executam a computação em si, temos um *nó de gerência* que serve como porta de entrada para o *cluster*. É nesse nó que os usuários realizam *login* e configuram as submissões para o sistema.

Cada *chassi* possui comutadores para as duas tecnologias (Infiniband e Ethernet), mas como um *chassi* suporta ao máximo 16 *blades*, mais dois *switches* Infiniband e um *switch* Ethernet são necessários para conectar os *chassis* e completar os 48 *blades* do *cluster*. É possível observar porque o custo da Infiniband é maior. As duas redes são conectadas à um nó de gerência, ao qual os *jobs* dos usuários são submetidos. Importante notar que o sistema de armazenamento (*storage*) é ligado apenas na rede de gerência, na qual é acessado através de sistema de arquivos distribuído. Um nó do *cluster* possui uma imagem própria do sistema operacional. Os sistemas operacionais mais utilizados são aqueles derivados do UNIX. Até o final da década de 1990, era comum que cada fabricante tivesse sua versão própria do UNIX. Com a popularização do Linux, os fabricantes encontraram

Figura 2.2: Topologia do Cluster do CENAPAD-UFC.



Fonte: Elaborada pelo autor.

um sistema UNIX robusto de baixo custo capaz de substituir as soluções proprietárias. O custo de manutenção do Linux é dividido por uma comunidade de vários desenvolvedores autônomos e empresas. Para os fabricantes de supercomputadores, resta desenvolver apenas *drivers* específicos e otimizações do *kernel* para sua máquinas. O *kernel* do Linux por si próprio não é o suficiente para um sistema funcional. Distribuições são coleções de aplicativos que formam um sistema completo para o usuário final. As distribuições Linux para supercomputadores apresentam customizações para executar aplicações de alto desempenho, mas são baseadas em distribuições já existentes no mercado. O *Red Hat Enterprise Linux* e o SUSE são escolhas populares.

Sistemas gerenciadores de recursos permitem que vários usuários utilizem um supercomputador com menor esforço administrativo e diminuição da ociosidade. Através de um sistema de filas, usuários submetem aplicações como tarefas no nó de gerência. Além de ponto de submissão, o nó de gerência também controla o acesso ao sistema de arquivos distribuído instalado no armazenamento. Na Listagem 2.1, temos um exemplo de um arquivo de definição de tarefa baseado no SLURM (*Simple Linux Utility for Resource Ma-*

nagement) (YOO; JETTE; GRONDONA, 2003) . Nesse arquivo, o usuário dá um nome a tarefa (CALCULO_PI), define a quantidade de nós de processamentos utilizados (4) e quantos processos são criados (48). Neste caso específico, são criados 12 processos em cada nó de processamento. O tempo máximo de execução da tarefa é pré-determinado (1 hora). Por último, uma linha definindo o nome e caminho arquivo binário a ser executado é informada (`./pi`).

Listagem 2.1: Arquivo de Definição de Tarefa

```
1 #!/bin/bash
2 #SBATCH -J CALCULO_PI
3 #SBATCH --partition particao_0
4 #SBATCH --nodes 4
5 #SBATCH --ntasks 48
6 #SBATCH --time 1:00:00
7
8 srun --resv-ports ./pi
```

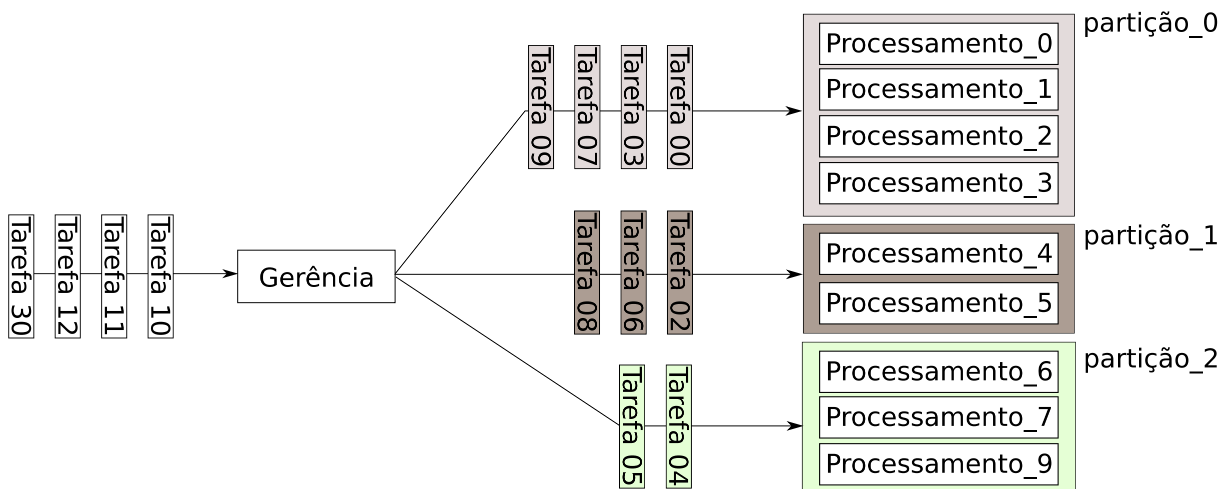
Ao ser submetido no nó de gerência, a tarefa entra numa fila inicial que analisa para qual partição ela está destinada. O conceito de partição corresponde a um subconjunto dos recursos do *cluster*. Essa divisão auxilia a definir prioridades. Por exemplo, um partição pode ser destinada apenas aos pesquisadores sênior de um instituição, incluindo todos os recursos disponíveis. Uma outra partição menor pode incluir apenas os recursos menos poderosos, destinada aos alunos de graduação. A Listagem 2.2 apresenta os parâmetros para definição de uma partição chamada *partição_0* no SLURM. Observa-se que além que configurar quais usuários têm acesso e quais nós de processamento fazem parte da partição, um limite máximo para o tempo de execução das aplicações é informado.

Listagem 2.2: Exemplo de Definição de Partição no SLURM

```
1 PartitionName=particao_0
2 Nodes=Processamento_[0-3]
3 Default=NO
4 MaxTime=240:00:00
5 AllowGroups=ufc,ufrgs,uespi,utfpr,ufpb,ufsc,ufma,ufpe,ufal
6 State=UP
```

A Figura 2.3 apresenta um exemplo esquemático do sistema de filas em gerenciador de recursos. O usuário submete tarefas ou *jobs* para um partição escolhida, de acordo com seus privilégios definidos pelo administrador. O sistema encaminha a tarefa para a fila adequada, na qual aguarda a execução. Uma vez que existem recursos computacionais livres, a tarefa é executada. O usuário não precisa monitorar todo o processo. O gerenciador de recursos se responsabiliza por todos os passos entre a submissão e a conclusão da execução.

Figura 2.3: Distribuição de filas em um gerenciador de recursos.



Fonte: Elaborada pelo autor.

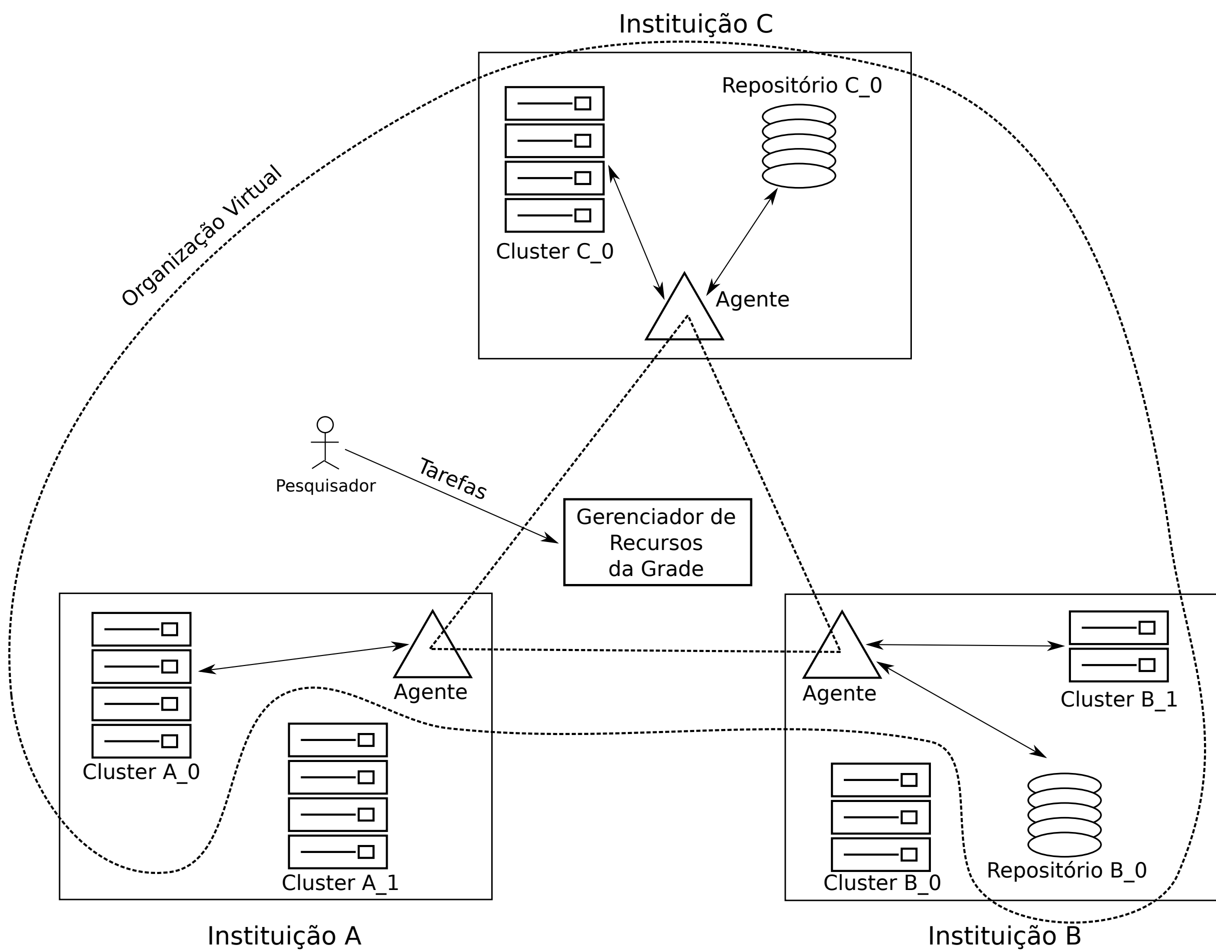
Apesar de controlar vários *blades* distintos, o gerenciador de recursos atua dentro de um mesmo domínio administrativo. O *cluster* é um sistema distribuído controlado. O acesso externo só é permitido ao nó de gerência, sendo que várias restrições de *firewall* são utilizadas para proteger o supercomputador de acesso indevido. Para permitir a utilização de *clusters* de instituições diferentes, outras tecnologias de computação distribuída mais amplas são necessárias, fornecendo controle de autorização e autenticação, além de um serviço de transferência de dados de alto desempenho.

2.1.2 Grades Computacionais

Em paralelo ao desenvolvimento do *hardware* para Computação de Alto Desempenho, no final da década de 1990, surgiram discussões sobre maneiras de fornecer acesso controlado à essas máquinas para um público mais amplo de pesquisadores e como otimizar o seu uso, em especial diminuindo os períodos de ociosidade. A princípio, essas máquinas eram adquiridas e administradas por grupos de pesquisa ou instituições independentes.

Como é natural no desenvolvimento científico, havia períodos de intensa utilização, intercalados por intervalos de análise dos resultados das simulações. Durante a análise, o supercomputador ficava a maior parte do tempo ocioso. Esses ciclos inutilizados poderiam muito bem ser aproveitados por outros pesquisadores. O que os cientistas também perceberam é que unindo seus recursos computacionais, problemas que antes exigiriam muito tempo para serem atacados individualmente seriam resolvidos de maneira mais rápida através da colaboração de diversos supercomputadores espalhados geograficamente. Dessa sinergia, surgiu o conceito de Grades Computacionais (FOSTER; KESSELMAN, 2003).

Figura 2.4: Esquema geral de uma grade computacional.



Fonte: Elaborada pelo autor.

A Figura 2.4 representa uma visão geral simplificada da arquitetura de uma grade computacional. Instituições com controle administrativo independente se unem em uma *organização virtual*, compartilhando recursos computacionais como *clusters*, repositórios de dados, largura de banda, equipamentos científicos, etc. O acordo de formação da grade define quanto cada instituição fornece de recursos e quanto cada uma tem direito de usufruir do conjunto da grade. Em cada membro, há um *agente da grade* que representa o de acesso aos recursos que estão disponibilizados. Cabe ao agente garantir o acesso controlado. Por exemplo, uma identidade de um usuário da grade deve ser mapeada para uma identidade local, com autorização limitada de acordo com os acordo de criação da grade.

Um usuário com permissão para utilizar a grade submete tarefas ao gerenciador de recursos da Grade. Esse serviço utiliza políticas e heurísticas definidas pelo acordo de formação da grade ao decidir para qual domínio encaminhará a requisição. Além das filas locais em cada *cluster*, a arquitetura da grade implica a criação de uma hierarquia de filas, formada pela fila global, as filas de cada agente e as filas dos próprios recursos. Apesar de estar representado na Figura 2.4 como uma entidade centralizada, o gerenciador de recursos da grade pode ser implementado de várias maneiras, inclusive como uma rede *peer to peer* formada pelos agentes.

Vários projetos de grades foram estabelecidos com sucesso (CATLETT, 2006) (ANDRADE, 2003) (SEGAL, 2000), contribuindo para o compartilhamento de recursos e a colaboração científica entre os participantes. Até a atualidade, a criação de grades computacionais continua sendo uma das melhores alternativas para fornecimento de serviços para Computação de Alto Desempenho.

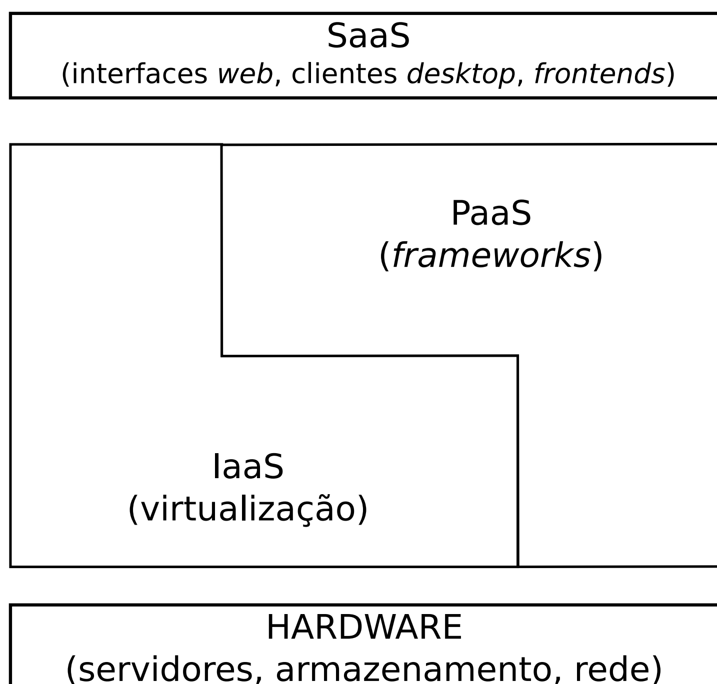
2.1.3 Nuvens Computacionais

As nuvens computacionais surgiram no início da década de 2000 como um modelo para permitir o acesso conveniente e sob demanda através da rede para um conjunto compartilhado de recursos computacionais configuráveis (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e liberados com um esforço mínimo de gerenciamento ou interação com o provedor de serviços (ZHANG; CHENG; BOUTABA, 2010).

A primeira função disseminada das nuvens foi fornecer servidores virtuais para a hospedagem de aplicações *web*. Essa categoria foi chamada de IaaS (*Infrastructure as a Service*) . Entretanto, o conceito se expandiu rapidamente, dando origem aos conceitos de PaaS (*Platform as a Service*) e SaaS (*Software as a Service*) que, assim como IaaS, buscam abstrair detalhes de infraestrutura do desenvolvedor e usuário final, respectivamente, mas

que não precisam necessariamente adotar virtualização. O mais importante é a noção de flexibilidade de configuração que o usuário percebe. Essa classificação pode ser visualizada na Figura 2.5.

Figura 2.5: Classificação das nuvens.



Fonte: Elaborada pelo autor.

Os serviços IaaS (*Infrastructure as a Service*) permitem alocar e desalocar máquinas virtuais com muita facilidade (através de uma API) e com pouco custo operacional. Um dos serviços de sucesso foi o Amazon EC2 (AMAZON, 2010). Esse serviço inovou com um sistema de cobrança baseado no tempo em que a máquina fica ativa, aceitando requisições. O custo por hora inicial era e ainda é muito competitivo em relação a serviços de hospedagem com máquinas físicas. Qualquer usuário com uma conta na Amazon pode criar suas instâncias, caracterizando assim uma *nuvem pública*. Apesar da oferta inicial ser customizada para aplicações *web*, não tardou para que pesquisadores considerassem unir várias instâncias de máquinas virtuais em um *cluster* virtual para a execução de aplicações científicas.

Dada a natureza do *datacenter* da Amazon e dos outros serviços de IaaS, nos quais um mesmo servidor físico hospeda várias máquinas virtuais (multi inquilino), que por sua vez se comunicam com outras máquinas virtuais através de uma rede compartilhada, a preocupação inicial era que a latência e banda da rede de um *cluster* virtual fossem inferiores as de um *cluster* computacional real. Essa ressalva se confirmou em experimentos iniciais (WALKER, 2008). Entretanto, artigos recentes (JORISSEN; VILA; REHR, 2012) demonstram que na atualidade já é possível executar aplicações científicas na nuvem com

pouca perda de desempenho se a aplicação tiver uma arquitetura compatível. Por exemplo, aplicações que consistem de tarefas independentes com pouca comunicação entre si executam com desempenho favorável nas nuvens.

Além do cenário de uma nuvem computacional pública como a Amazon EC2, outro modelo de implantação possível determina que uma instituição pode configurar uma *nuvem privada* em execução num *datacenter* ou *cluster computacional* próprio. A princípio não há vantagem aparente, pois adicionar uma camada de virtualização acarreta uma sobrecarga. Mas quando se considera a execução de aplicações com perfil de configuração conflitantes (sistemas operacionais diferentes, versões distintas da linguagens, bibliotecas incompatíveis, etc), o uso de uma nuvem privada permite a utilização de um mesmo recurso físico para tais aplicações. Pacotes como o OpenStack (LEDYAYEV; RICHTER, 2014) e o OpenNebula (RUIVO, 2014) são indicados para esse cenário.

De acordo com o NIST (*National Institute of Standards and Technology*), elasticidade em nuvens é a alocação e liberação dinâmica de recursos sem a intervenção direta do usuário final. A abstração resultante é a ilusão de um conjunto de recursos infinito que podem ser adquiridos e liberados a qualquer momento durante a execução (MELL; GRANCE, 2011). Uma característica importante das nuvens é fornecer meios para que o usuário ou desenvolvedor estabeleçam regras para o provisionamento automático de recursos durante o ciclo de vida da aplicação. O caso clássico para a elasticidade é um servidor *web* que aloca mais recursos a medida que suas páginas são acessadas com mais frequência. Quando o número de acessos diminui, a elasticidade permite desalocar os recursos adquiridos.

A elasticidade fornecida pelas nuvens é um característica importante para aplicações escaláveis. No contexto de nuvens, a escalabilidade é a capacidade de uma aplicação fornecer melhor tempo de resposta ou lidar com cargas de trabalho maiores após a adição de novos recursos (GALANTE; BONA, 2012). Durante seu ciclo de execução, uma aplicação escalável pode requisitar mais recursos para uma nuvem elástica sem a intervenção do usuário ou administrador. A escalabilidade trata o aumento do número de recursos, mas o caso inverso, no qual uma aplicação libera recursos quando não são mais necessários, também é importante para o controle do custo de execução. Iremos apresentar a definição de escalabilidade para Computação de Alto Desempenho na Seção 2.4.2, mas adiantamos que sistemas paralelos escaláveis também relacionam a carga que deve ser processada e os recursos disponíveis.

Da maneira que foi apresentada, a elasticidade é perceptível ao administrador de nuvens IaaS ou ao desenvolvedor que faz uso de nuvens IaaS ou PaaS. Essa interação não ocorre de maneira direta, exigindo monitoramento constante por parte do elemento humano. Regras e políticas são definidas para ditar o comportamento do sistema e aplicação

no caso de alteração dos recursos. Ao usuário final, o que resta é a noção de um sistema executando com a qualidade de serviço desejada e custo financeiro reduzido. No Capítulo 4, apresentamos uma discussão mais detalhada do uso de nuvens para Computação de Alto Desempenho.

2.1.4 Nuvens versus Grades

Finalizamos a seção com uma comparação entre grades e nuvens. A grade é uma solução que tem como alicerce o conceito de cooperação através de uma organização virtual, com foco na integração de recursos. Não há sentido em uma grade na qual os recursos pertencem à uma mesma entidade e estão sujeitos às mesmas políticas de acesso e utilização. Nas nuvens os recursos em geral pertencem a uma mesma organização, que escolhe oferecer acesso público ou privado a esses recursos através de serviços.

O acordo de participação na grade em geral não envolve recursos financeiros. Em outras palavras, um pesquisador não paga para submeter suas aplicações à grade. A definição da qualidade de serviço entre as diversas submissões é feita através de uma política geral. Uma possibilidade é adotar uma rede de troca de favores: os usuários da instituição que forneceu a maior quantidade de horas de processamento tem maior prioridade do que usuários de outras organizações. Outra opção é definir uma ordem de prioridade fixa entre as instituições. As nuvens públicas oferecem recursos de acordo com um preço tabelado por hora de execução. Existem vários perfis de utilização de acordo com as necessidades de cada aplicação. Por exemplo, aplicações que dependem da memória podem optar por uma configuração com poucos núcleos e quantidade considerável de memória por nó. O preço por hora desse tipo de instância é diferente de uma configuração que fornece mais núcleos. Portanto, a nuvem oferece maior flexibilidade em troca de um investimento financeiro.

O desempenho das aplicações na grade é equivalente à execução do código em um dos *clusters* que compõem a organização virtual. Não há uma camada de virtualização, a execução é direta sobre a máquina alocada. Entretanto, para ter acesso aos *clusters*, a aplicação deve ter requisições alocadas pelo gerenciador de cursos, o que leva a um intervalo de espera variável na fila. A nuvem impõe sobrecarga de virtualização que tem impacto negativo no desempenho, porém a alocação dos recursos é imediata, existindo apenas o tempo constante de criação das máquinas virtuais.

Para o usuário final, mesmo em serviços IaaS, a nuvem apresenta um nível maior de abstração. Apesar de ter que configurar as instâncias, o usuário não tem contato com os detalhes da infraestrutura do conjunto de *datacenters* que hospedam a nuvem. E soluções de mais alto nível como PaaS e SaaS tornam transparente a própria configuração das

máquinas virtuais, permitindo ao desenvolvedor se concentrar no tratamento de dados e organização dos algoritmos. Na grade, apesar da existência do gerenciador de recursos, o usuário é exposto aos detalhes da configuração de cada *cluster*. Existem portais *web* para facilitar o acesso, mas essas ferramentas apenas evitam o uso da linha de comando, ainda exigem configuração cuidadosa das submissões para obter bom desempenho.

Apesar das diferenças citadas, para a Computação de Alto Desempenho, ambos os conceitos, grades e nuvens, apresentam formas diferentes de alcançar a mesma finalidade: o acesso a um conjunto de recursos computacionais mais amplo e potencialmente distribuído geograficamente.

2.2 Modelos de Programação

Na Seção 2.1.1, apresentamos a arquitetura de *clusters* com dois níveis de organização de memória: memória compartilhada dentro de um nó ou memória distribuída entre os nós. A organização da memória tem impacto direto no modelo de programação, logo os dois modelos de programação predominantes para Computação de Alto Desempenho são os de *memória compartilhada* e *troca de Mensagens*.

2.2.1 Modelo de Variáveis Compartilhadas

No modelo de variáveis compartilhadas, tarefas concorrentes compartilham um espaço de endereçamento comum, no qual elas podem escrever e ler de maneira assíncrona. Essas tarefas podem ser representadas por processos ou *threads*. Sendo as tarefas processos, como no caso de arquiteturas NUMA (*Non Uniform Memory Access*), uma biblioteca pode ser utilizada para mapear regiões compartilhadas da memória entre processos diferentes. Por exemplo, na biblioteca POSIX, esse mapeamento resulta em um arquivo virtual que representa a região compartilhada. O sistema operacional deve oferecer mecanismos de travas ou semáforos para o controle de concorrência.

Na atualidade, o modelo mais difundido de programação por variáveis compartilhadas é através do uso de *threads*[1] como tarefas. Nesse modelo, dentro do sistema operacional, uma abstração de processo pode abrigar vários fluxos de execução concorrentes (TANENBAUM, 1992). Cada fluxo é chamado de *thread*. Os *threads* de um processo compartilham a memória do mesmo, além de outros recursos como arquivos abertos. Entretanto, cada *thread* também possui informações locais. Para controlar o acesso a regiões de memória compartilhadas a estrutura de dados semáforo é a mais utilizada, com operações que

[1] Também conhecidas como *processos leves*, em português, muito embora o uso desse termo não tenha se tornado muito difundido ao longo dos anos.

permitem bloquear ou liberar *threads* concorrentes, a fim de implementar com mais simplicidade protocolos de exclusão mútua e sincronização condicional.

A programação nesse modelo pode ser feita de duas maneiras: através de uma biblioteca ou utilizando diretivas de compilador. Com uma biblioteca, uma interface permite ao programador invocar funções para criar *threads*, designar qual código elas executaram e tratar questões de escalonamento e sincronização. Dessa forma, cabe ao desenvolvedor controlar o ciclo de vida de fluxo de execução. A biblioteca POSIX *Threads* (MUELLER et al., 1993) foi resultado de um esforço de padronização da indústria e academia, constituindo atualmente a solução de suporte a programação com *threads* encontrada na maioria dos sistemas UNIX.

Através do uso de diretivas de compilação, o desenvolvedor, em teoria, pode adaptar facilmente um código serial para paralelo. Ele é liberado do controle direto de cada *thread*, apenas informa quais porções deseja paralelizar de acordo com padrões pré-definidos. A princípio, pode parecer uma solução pouco flexível, mas observa-se na prática que trechos de código específicos como laços de iteração são adequados para execução paralela. Por vários anos, cada fabricante desenvolvia sua versão de diretivas para compiladores. No final da década de 1990 um consórcio foi formado e o resultado foi o padrão *OpenMP* (DAGUM; MENON, 1998), disponível na atualidade para a maioria dos compiladores.

2.2.2 Modelo de Troca de Mensagens

No paradigma de troca de mensagens, a execução de uma aplicação científica é intercalada entre períodos de computação e comunicação. Durante a computação, várias tarefas ou processos executam acessando apenas a memória local da máquina na qual residem. Após a fatia local dos cálculos ser concluída, na fase de comunicação, tarefas em máquinas diferentes trocam mensagens para sincronização e nova divisão do trabalho restante. A comunicação é feita através de chamadas a funções de uma biblioteca. Em geral, a transferência de dados requer cooperação entre tarefas para ocorrer. Por exemplo, uma operação para enviar dados, invocada por um processo, precisa da invocação de uma operação correspondente em outro processo para recebê-los. A comunicação também pode ocorrer envolvendo várias tarefas invocando uma operação coletiva de troca de dados.

Assim como no caso do OpenMP, originalmente cada fabricante oferecia sua interface para a biblioteca de troca de mensagens. No começo da década de 1990, um consórcio definiu o padrão MPI (*Message Passing Interface*) (GROPP, 1996) . Como a sigla já informa, a padronização ocorreu a nível de interface, ou seja, a assinatura das funções que fazem parte de uma distribuição MPI. Os detalhes internos da implementação podem variar, de modo que cada fabricante realiza otimizações de acordo com a rede de interco-

nexão ou barramento do supercomputador. Mesmo com diferenças internas, a interface padrão elevou a portabilidade dos programas paralelos.

Como foi visto na Seção 2.1, um supercomputador moderno é um conjunto de máquinas com memória compartilhada interligadas por uma rede. Nesse cenário, aplicações são desenvolvidas usando os dois modelos de programação apresentados. Dentro um nó de processamento há *threads* (criadas por diretivas OpenMP ou chamadas POSIX) que se comunicam com *threads* em outras máquinas através de troca de mensagens (chamadas MPI). Esse modelo híbrido apresenta desafios para o controle de concorrência, mas ao mesmo tempo, possibilita extrair o máximo do desempenho das plataformas modernas.

2.3 Tipos de Tarefas Paralelas

Como discutimos na Seção 2.1, as plataformas de Computação de Alto Desempenho de uso compartilhado utilizam gerenciadores de recursos para organizar o acesso dos usuários. Nesta seção, discutimos os tipos de programas paralelos sob a ótica do gerenciador de recursos (FEITELSON; RUDOLPH, 1996).

2.3.1 Objetivos do Escalonamento

Além de reforçar políticas de acesso e segurança, uma tarefa importante do gerenciador de recursos é realizar o escalonamento, ou seja, escolher qual plataforma ou recursos são utilizados para atender a próxima tarefa na fila. O principal objetivo do escalonamento é garantir que qualquer tarefa tenha a chance de executar sua próxima instrução sempre que encontre-se apta para tal. Quando isso não ocorre, ou seja, dizemos que o código não é livre de *inanição* (*starvation*). Apesar de óbvio, a busca por garantir outras métricas pode colocar em perigo a execução de tarefas de pouca prioridade.

Outro objetivo claro é otimizar a *taxa de utilização* dos recursos. Essa métrica está ligada com a carga do sistema: se a fila não for grande e todas as tarefas puderem ser atendidas ao mesmo tempo, a taxa de utilização será igual a soma da carga das tarefas. Uma métrica também relacionada com a carga é a *vazão*, ou número de tarefas por unidade de tempo. Perceba que otimizar a taxa de utilização não é o mesmo que otimizar a vazão: executar tarefas com alto paralelismo favorece a utilização, enquanto tarefas com baixo paralelismo favorece a vazão.

As taxas mencionadas até agora são importantes para os administradores das plataformas, mas o usuário final está preocupado com a execução do seu conjunto de tarefas, não com o sistema como um todo. Para quem submete tarefas, o objetivo mais importante

é reduzir o *tempo médio de resposta*. Esse intervalo é definido como o tempo decorrido desde a submissão até o final da execução da tarefa. Atribuir a cada usuário uma fatia de tempo adequada está relacionado ao conceito de *justiça* (*fairness*).

2.3.2 Classificação

Apesar de até agora mencionarmos tarefas como um conjunto homogêneo, na realidade existem diversos tipos de tarefas. Uma primeira possível divisão é entre tarefas que executam programas seriais e tarefas que representam programas paralelos. Dado o escopo do nosso trabalho, estamos interessados em discutir o segundo tipo. Uma classificação importante para tarefas paralelas é de acordo com o conjunto de recursos que utilizam no decorrer na execução. A Tabela 2.1 apresenta uma classificação de acordo com a quantidade de recursos (nós de processamento) utilizada.

Tabela 2.1: Classificação de tarefas paralelas de acordo com quantidade nós de processamento utilizada.

Quem decide a quantidade?	**quando é decidido?**	
	estático	**dinâmico**
usuário	rígido	evolutivo
sistema	moldável	maleável

Fonte: Elaborada pelo autor.

Tarefas *rígidas* precisam de um número determinado de nós de processamento para executar, especificado na submissão (entrada na fila). Esse tipo de tarefa não executa em uma quantidade menor de recursos e nem faz uso de mais nós de processamento. Do ponto de vista do programador, escrever um programa para executar em uma tarefa rígida faz sentido em casos nos quais a decomposição do algoritmo é ótima em certo número de processadores (maiores detalhes na Seção 2.4).

No outro extremo de flexibilidade, tarefas *evolutivas* podem mudar o conjunto de recursos durante a execução. O programa em execução na tarefa é o responsável por iniciar qualquer alteração nos recursos, fazendo requisições ao gerenciador. Esse padrão é comum em programas paralelos projetados em fases com grau de paralelismo variável. Com o advento da elasticidade nas nuvens, tarefas paralelas ganham importância pois podem aproveitar novos recursos disponíveis na nuvem.

No cenário atual de um ambiente de CAD, o tipo mais comum de tarefas são as *moldáveis*. Uma tarefa *moldável* permite que o número de nós de processamento seja configurado no início da execução, sendo que o programa se adapta a esse número que se

mantém o mesmo durante toda a execução. A escolha da quantidade de recursos pode ser determinada pelo usuário na submissão, mas existe a opção de permitir que o gerenciador configure esse número de acordo com a disponibilidade de recursos. O último caso é útil quando o usuário deseja que a execução comece o mais rápido possível.

Tarefas *maleáveis*, assim como as evolutivas, permitem adaptar o número de nós de processamento durante a execução. Entretanto, a decisão de adaptação não parte do programa em execução na tarefa, mas sim do gerenciador de recursos. Também é um cenário importante para a elasticidade nas nuvens, mas como a decisão parte do sistema, esse tipo de tarefa é mais natural quando se deseja otimizar os requisitos do administrador (utilização e vazão) do que do usuário final (tempo de resposta).

2.4 Escalabilidade de Programas Paralelos

Podemos afirmar que, para um programa paralelo, quanto maior a quantidade de recursos melhor será o desempenho? Infelizmente, a resposta nem sempre é afirmativa. Ao contrário de um programa serial, no qual quanto maior o *clock* do processador mais rápida será a execução, no programa paralelo existem sobrecargas que impõem um limite à escalabilidade. A principal limitação é a comunicação entre os elementos de processamento paralelos (sejam processadores ou núcleos). Tal restrição tem impacto nos supercomputadores modernos, que apesar de apresentarem memória compartilhada dentro de um nó de processamento, utiliza troca de mensagens através da rede de computação do *cluster*.

A relação entre o intervalo útil de execução da computação e o tempo gasto em sincronização e transferência de dados é um fator importante para decidir se a adição de novos recursos é benéfica ou não. Como o peso da comunicação varia de acordo com a plataforma, um algoritmo paralelo não pode ser avaliado isolado da arquitetura paralela na qual executa. Um sistema paralelo é a combinação de um algoritmo e arquitetura paralela. Nesta seção, apresentamos a modelagem analítica de programas paralelos com o objetivo de demonstrar o impacto da elasticidade das nuvens no desempenho da aplicação. A discussão aqui apresentada é baseada em (KUMAR, 1994).

Dentre as origens da sobrecarga em programas paralelos, já citamos a *comunicação entre os elementos de processamento* como um dos fatores preponderantes. Outro fator a se considerar é a *ociosidade*, que pode ser decorrente da sincronização ou do balanceamento inadequado da carga entre os elementos de processamento. No caso do desbalanceamento, um elemento de processamento que finalizar suas tarefas primeiro que os outros componentes do sistema paralelo precisa esperar que todos terminem para prosseguir para a próxima etapa.

Outra situação de sobrecarga é a *computação em excesso* para permitir a paralelização. Não é raro o caso no qual o melhor algoritmo serial não pode ser paralelizado, forçando a utilização de um algoritmo pior para a versão paralela, que acaba por realizar mais trabalho. Para elicitar as sobrecargas de sistemas paralelos, podemos estudar a arquitetura do *hardware* em alto nível de detalhes, equalizando todos os atrasos decorrentes da rede de interconexão, barramentos internos, etc. Entretanto, essa abordagem, além de difícil execução, envolve detalhes que não são da natureza do algoritmo, como a topologia da rede. Uma abordagem mais tratável é observar a diferença nos intervalos de execução paralelo e serial e abstrair-se da sobrecarga total a partir de métricas de desempenho.

2.4.1 Métricas de Desempenho

Precisamos definir o desempenho de uma aplicação de Computação de Alto Desempenho para entender até que ponto sua escalabilidade afeta o resultado final. A métrica óbvia para avaliar o desempenho é o *tempo de execução*. O tempo de execução paralela é o intervalo entre o início e o momento em que o último processo termina a execução. No restante do texto, o tempo de execução serial é representado por T_S, e o paralelo T_P.

As sobrecargas (*overheads*) enfrentadas pela execução paralela são abstraídas em uma equação chamada *função de overhead*, representada pelo símbolo T_O. A sobrecarga de um sistema paralelo é o tempo total gasto por todos os elementos de processamento menos o tempo de execução do melhor algoritmo serial, dada uma mesma entrada. Sendo p o número de elementos de processamento, o tempo total gasto é pT_P. A função é definida de acordo com a Equação 2.1.

$$T_O = pT_P - T_S \tag{2.1}$$

A principal razão para se adotar paralelismo é obter um tempo de execução menor do que a versão serial do programa. Para medir esse ganho, temos a métrica *speedup*, que é definida como a média entre o tempo levado para resolver o problema utilizando apenas um único elemento de processamento em relação ao intervalo necessário para resolver o mesmo problema em um computador paralelo com p elementos de processamento idênticos (Equação 2.2). É importante notar que T_S toma como base o melhor algoritmo serial disponível, que não é necessariamente o mesmo que serve como base para a versão paralela.

$$S = \frac{T_S}{T_P} \tag{2.2}$$

O ideal seria que o *speedup* fosse sempre igual a p. Em outras palavras, cada elemento de processamento adicionado, seja o primeiro ou centésimo, traria o mesmo ganho de

desempenho. Mas, na realidade, a maioria dos programas paralelos não apresenta essa qualidade. A métrica para medir esse cenário é a *eficiência*, que busca medir a fração de tempo útil de um elemento de processamento. Ela é definida como a razão entre o *speedup* e o número de elementos de processamento (Equação 2.3).

$$E = \frac{S}{p} \tag{2.3}$$

A última métrica que vamos apresentar é o *custo*. Temos que ter cuidado para não fazer confusão com o custo financeiro das nuvens IaaS, mas há relação entre os termos. O custo de resolver um problema em um sistema paralelo é o produto do tempo de execução paralelo e o número de elementos de processamento. Trata-se do primeiro termo da Equação 2.1, dado por pT_P. Dizemos que um sistema paralelo é de *custo ótimo* se o custo de resolvê-lo em computador paralelo tem o mesmo crescimento assimptótico em função da entrada que o melhor algoritmo serial. A granularidade da decomposição tem impacto na determinação se um algoritmo é ou não de custo ótimo. Considere uma situação em que há uma grande quantidade disponível de elementos de processamento para resolver um problema com determinada entrada fixa. Uma granularidade fina (muitas tarefas) pode acarretar várias etapas de comunicação, aumentando a sobrecarga. Para garantir o custo ótimo, a decomposição do algoritmo deve buscar usar o máximo de recursos sem que a sobrecarga prejudique o desempenho. A determinação do ponto ideal é o que nos levar a estudar a escalabilidade de sistemas paralelos.

2.4.2 Escalabilidade de Sistemas Paralelos

Podemos utilizar as equações da Seção 2.4.1 para reescrever a eficiência em função da sobrecarga e do tempo de execução serial.

$$E = \frac{S}{p} = \frac{T_S}{pT_P} \tag{2.4}$$

$$E = \frac{1}{1 + \frac{T_O}{T_S}} \tag{2.5}$$

Quanto maior o valor de p, maior o valor de T_O (Equação 2.1). No melhor dos casos, a relação é linear. Entretanto, T_O pode ter complexidade maior que a linear devido às sobrecargas. Podemos afirmar que, para todos os programas paralelos, mantendo o tamanho do problema fixo (ou seja, o tempo serial T_S constante), aumentando o número de elementos de processamento, aumentamos também T_O. E pela Equação 2.5, a eficiência diminui. Se o tamanho do problema for incrementado junto com p, é possível manter a

eficiência constante. Um sistema paralelo que permite manter a eficiência constante ao incrementar simultaneamente o número de elementos de processamentos e o tamanho do problema é um *sistema escalável*.

Devemos enfatizar a relação entre o tamanho do problema e o número de elementos de processamento na definição de escalabilidade. Salvo casos específicos como aplicações de tempo real ou interativas, programas de Computação de Alto Desempenho são submetidos para execução nas plataformas descritas na Seção 2.1 junto com a entrada já definida. Em outras palavras, durante a execução, o tamanho do problema não varia. Para determinar se precisamos adicionar ou remover recursos, precisamos saber o quão distantes estamos do ponto ideal que fornecer a melhor eficiência (e por consequência, o melhor desempenho). Portanto, há a necessidade de uma equação que relacione o tamanho do problema, a eficiência e o número de elementos de processamento.

Até agora não fornecemos uma definição precisa do tamanho do problema. A solução é adotar o tamanho da entrada como tamanho do problema. O problema dessa abordagem é que as versões serial e paralela podem utilizar estruturas de dados diferentes, sendo que o tamanho da entrada para uma versão é incompatível com o da outra. Uma definição consistente com computação paralela é adotar o tamanho do problema como a quantidade total de operações básicas (sem operações de comunicação ou sincronização) necessárias para resolver o problema. Para chegar ao número de operações, observamos o melhor algoritmo sequencial disponível. O tamanho do problema é definido em termos da complexidade sequencial, que por sua vez guarda relação com o tamanho da entrada. Para simplificar os cálculos, assumimos que cada operação básica leva uma unidade de tempo. Dessa forma, o tamanho do problema pode ser dado por uma função linear em T_S. Considerando o tamanho do problema como W e sabendo que o custo da execução é número de operações básicas mais a sobrecarga, a Equação 2.6 fornece o tempo de execução paralelo em função das outras grandezas.

$$T_P = \frac{W + T_O(W, p)}{p} \tag{2.6}$$

Substituindo a Equação 2.6 nas Equações 2.2 e 2.3, chegamos nas Equações 2.7 e 2.8.

$$S = \frac{Wp}{W + T_O(W, p)} \tag{2.7}$$

$$E = \frac{1}{1 + \frac{T_O(W,p)}{W}} \tag{2.8}$$

A partir da Equação 2.8, concluímos que, dependendo do sistema paralelo (e suas sobrecargas), a taxa de crescimento de W em relação à p para manutenção da eficiência

varia. Em alguns casos, W pode precisar de crescimento exponencial em relação a p para manter a eficiência. Esses sistemas seriam pouco escaláveis. Já nos casos em que o crescimento é linear, o sistema é altamente escalável. Adotando $K = \frac{E}{1-E}$, temos a Equação 2.9 como a função de *isoeficiência* de um sistema paralelo.

$$\frac{T_O(W,p)}{W} = \frac{1-E}{E}$$
$$W = \frac{E}{1-E} T_O(w,P)$$

$$W = KT_O(W,P) \tag{2.9}$$

A função de isoeficiência auxilia na escolha do número de recursos na submissão dos programas paralelos. É uma aproximação, mas se o desenvolvedor conseguir relacionar a sobrecarga com o tamanho do problema, poderá evitar o desperdício de recursos ou plataformas que aumentem a sobrecarga. Entretanto, a isoeficiência não é suficiente para responder a seguinte pergunta: dada um tamanho de problema fixo, em quanto posso aumentar o número de elementos de processamento e ainda esperar uma diminuição do tempo de execução paralelo? Para responder a essa pergunta, precisamos definir T_P para um dado W fixo em função de p. Igualando a diferencial de T_P em função de p à zero, temos o valor de p no qual o tempo de execução é mínimo.

$$\frac{\mathrm{d}}{\mathrm{d}p} T_p = 0 \tag{2.10}$$

A questão de definir T_P para um W pode ser resolvida através de análise, mas a função de isoeficiência pode guiar esse estudo através da execução em pequenas instâncias, pois os valores do *speedup* e eficiência podem ser obtidos experimentalmente. Vale ressaltar que o fato de que o tempo mínimo em função da entrada pode ser menor que a função de isoeficiência assintoticamente.

2.5 Conclusões

O objetivo deste capítulo foi apresentar um panorama da Computação de Alto Desempenho. Considerando o que foi apresentado sobre as infraestruturas e da escalabilidade de algoritmos paralelos, podemos concluir que aplicações com sobrecarga significativa de comunicação não são boas candidatas para nuvens. A multi hospedagem de máquinas

virtuais na nuvem também causa problemas para aplicações paralelas sensíveis ao uso inadequado da memória cache. Entretanto, apesar do cenário inicialmente desfavorável, nuvens fornecem recursos sob demanda e têm suporte à elasticidade. Os supercomputadores (*clusters*) e as grades são infraestruturas ideais para Computação de Alto Desempenho, porém raramente são de uso exclusivo e apresentam tempo de fila e limite para tempo total de execução. Habilitando a reconfiguração elástica de aplicações, podemos diminuir a diferença entre as infraestruturas na garantia da qualidade de serviço.

3. Computação de Alto Desempenho Baseada em Componentes (CBHPC)

Nas últimas décadas, a complexidade de aplicações de Computação de Alto Desempenho cresceu a medida que pesquisadores começaram a tratar problemas envolvendo vários modelos de diferentes áreas de conhecimento. Devido à natureza multidisciplinar dessas aplicações, as equipes de desenvolvimento aumentaram em número de integrantes, passando a incluir membros de diversas instituições. Essa colaboração, enquanto necessária para o progresso da ciência, introduz novos desafios para o processo de desenvolvimento de *software* científico. É natural que pesquisadores com formações distintas apresentem estilos de programação diferentes e abordagens diversas para os mesmos problemas.

Em paralelo ao cenário mais diverso dentre os profissionais da área, a Computação de Alto Desempenho caminha para uma maior heterogeneidade em relação ao *hardware*. Como discutido no Capítulo 2, os supercomputadores apresentam aceleradores e processadores com arquiteturas heterogêneas, levando a utilização de vários modelos de programação em uma mesma aplicação. Esses desafios deixam claro que a comunidade de Computação de Alto Desempenho precisa de uma abordagem para desenvolvimento de *software* que facilite a gerência de complexidade ao mesmo tempo que mantém desempenho escalável e eficiente do código (MCINNES, 2006a). O ideal seria um cenário em que um cientista pode focar em desenvolver apenas o código do experimento que é de interesse primário para sua pesquisa, reutilizando código testado já existente desenvolvido por especialistas nos outros aspectos computacionais da simulação.

A engenharia de *software* baseada em componentes (no inglês, *Component-Based Sofware Engineering* - CBSE) (WANG; QIAN, 2005) busca facilitar a gerência de complexidade ao permitir que blocos de código desenvolvidos por grupos diferentes consigam interagir em uma mesma aplicação, que é vista como um grupo de componentes que interagem entre si apenas através de interfaces bem definidas, sustentados por um ambiente de execução governado por um *framework*. Nesse contexto, *componentes* são unidades de funcionalidade de código que em conjunto apresentam a função final da aplicação. Eles podem ser vistos como *caixas-pretas*, sendo que a compreensão da lógica interna não é necessária para o reuso, mas apenas o conhecimento de suas interfaces. Várias implementações diferentes de uma mesma funcionalidade podem oferecer a mesma interface, sendo interoperáveis. Entretanto, sua lógica interna pode diferir para acomodar diferentes aspectos não funcionais, tais como características de desempenho e estilos de programação.

Neste capítulo, apresentamos dois esforços para modelos de plataformas de componentes voltadas às necessidades de aplicações de Computação de Alto Desempenho. Na Seção 3.1, discutimos a arquitetura CCA baseada no conceito de portas e na distribuição de um mesmo componente em vários processos distintos. O GCM é apresentado na Seção 3.2, trata-se de uma extensão de modelos pré-existentes para prover arquiteturas baseadas em componentes para grades computacionais. Ao final de cada seção, soluções de como fornecer reconfiguração e adaptação a aplicações desenvolvidas em cada modelo são detalhadas.

3.1 CCA

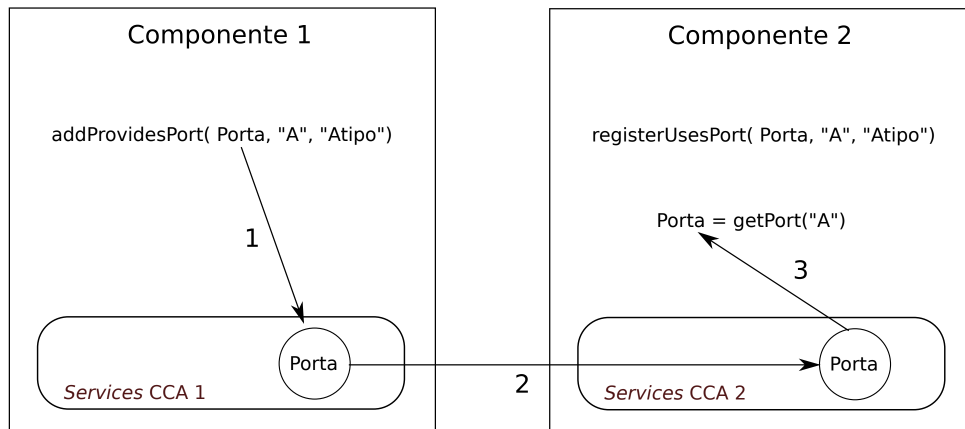
CCA (*Common Component Architecture*) tem como objetivo definir um modelo de componentes adequado para a computação científica, enfatizando os requisitos de desempenho e escalabilidade que não são priorizados nos *frameworks* comerciais baseados em componentes (ARMSTRONG, 2006). A CCA apresenta um modelo concreto, mas não define uma técnica única para ligação de código escrito em linguagens diferentes. Essa definição fica a cargo dos desenvolvedores dos *frameworks* compatíveis com o modelo. Entretanto, a CCA reforça um mecanismo de conexão direta, ou seja, chamadas de função são realizadas diretamente entre os módulos, sem código intermediário que possa afetar o desempenho.

3.1.1 O Padrão de Projeto *Uses/Provides*

O principal conceito para entender como é feita a interação entre dois componentes na CCA são as *portas*. Na orientação a objetos em uma linguagem como Java, uma interface apenas define os métodos que o módulo ou classe que a implementa oferece para os outros integrantes do sistema. Ao definir portas como interfaces abstratas, um componente pode não só detalhar quais funcionalidades provê ao sistema, mas também quais procedimentos ou recursos deseja utilizar de outros componentes. Componentes podem prover portas (*provides ports*), implementando funcionalidades, ou são usuários de portas (*uses ports*), realizando chamadas em uma porta fornecida por outro componente. A interface de um componente é definida pelo conjunto dos dois tipos de portas.

Um *framework* compatível com a CCA deve realizar a ligação entre as portas *uses* e *provides* das interfaces dos componentes. Para tal, ele deve invocar um método *setServices* em cada componente criado sob seu controle. Esse método registra dentro das estruturas de dados do *framework* referências aos objetos instanciados que representam as portas *provides* dos componentes, além de registrar as portas *uses* que cada componente precisa.

Figura 3.1: Padrão de projeto *uses/provides*.



Fonte: (ARMSTRONG, 2006).

A Figura 3.1 apresenta o fluxo de registro e utilização de portas. Cada componente informa seus requisitos de portas e o *framework* trata de transferir as referências aos objetos que as implementam. Quando todos os componentes compartilham o mesmo espaço de endereçamento, a invocação de um método em uma porta é feita de forma direta. Entretanto, não há nada na especificação que impeça um *framework* de permitir invocação remota de métodos.

Listagem 3.1: Componente `HelloWordServer`.

```
1  import gov.cca
2
3  class StringProducerPort(gov.cca.Port):
4      def sayHello(self):
5          print "Hello World"
6
7  class Component(gov.cca.Component):
8      def __init__(self):
9          self.stringProducerPort =
10             StringProducerPort("HelloServer.StringProducerPort")
11         return
12     def setServices(self, services):
13         self.services = services
14         services.addProvidesPort(self.stringProducerPort, "HelloServer",
15                                  "HelloServer.StringProducerPort", None)
16         return
```

A Listagem 3.1 na linguagem Python apresenta um componente servidor que define uma porta com um único método (`sayHello`) e a exporta sob a nomenclatura `HelloServer`. O método `setServices` trata de registrar essa porta no *framework* que cria o componente. Nesse registro, o componente cadastrado informa a referência ao objeto

que representa a porta, o nome da porta e seu tipo. Com essas informações disponíveis, outros componentes podem acessar os métodos da porta.

Um exemplo para um componente cliente está na Listagem 3.2. Ele define uma porta *provides* (`HelloClientGoPort`) que estende `gov.cca.ports.GoPort`. Essa é uma porta especial que será chamada pela aplicação para iniciar a execução. No método `go` dessa porta, vemos que o componente cliente recupera a porta `StringProducerPort` registrada sob o nome `HelloServer` pelo componente servidor e invoca o único método que ela possui. No método `setServices`, além de registrar sua `GoPort` para a aplicação, o cliente também precisa registrar que necessita da porta `HelloServer` do servidor através do método `registerUsesPort`.

Listagem 3.2: Componente HelloWordClient.

```
1  import gov.cca
2  import gov.cca.ports
3
4  class HelloClientGoPort(gov.cca.ports.GoPort):
5      def __init__(self, component):
6          self.component = component
7          super(HelloClientGoPort, self).__init__()
8          return
9      def go(self):
10         port = self.component.services.getPort("HelloServer")
11         port.sayHello()
12         self.component.services.releasePort("HelloServer")
13         return
14
15 class Component(gov.cca.Component):
16     def __init__(self):
17         self.goPort = HelloClientGoPort(self)
18         return
19     def setServices(self, services):
20         self.services = services
21         services.registerUsesPort("HelloServer",
22                                   "HelloServer.StringProducerPort",
23                                   None)
24         services.addProvidesPort(self.goPort,
25                                  "GoPort",
26                                  "gov.cca.ports.GoPort",
27                                  None)
28         return
```

Com os componentes e portas definidos, na Listagem 3.3 temos a aplicação que faz a ligação. Inicialmente, uma instância do *framework* é criada e através dela uma instância de cada componente é definida. Dentro do método createInstance, o *framework*

define uma estrutura de dados para armazenar as informações sobre as portas (services) e invoca o método setServices de cada componente criado passando uma referência para essa estrutura como parâmetro. O método connect realiza a ligação entre as portas dos componentes, transferindo referências para os objetos que representam as portas. Por fim, a própria aplicação recupera a GoPort do complete cliente e dá início a execução.

Nas listagens apresentadas, não há explicação sobre a criação e alocação de processos para execução da aplicação. Nesse exemplo, a execução é serial, o modelo de componentes reorganiza o código para facilitar o reuso, mas para a computação científica o paralelismo é importante.

Listagem 3.3: Aplicação conectando os componentes.

```
1  from framework.manage.builders import FrameworkHandle
2
3  fwk = FrameworkHandle()
4  server = fwk.createInstance("HelloServerInstance",
5                               "HelloServer.Component",
6                                None)
7  client = fwk.createInstance("HelloClientInstance",
8                               "HelloClient.Component",
9                                None)
10 fwk.connect(client, "HelloServer", server, "HelloServer")
11 goport = fwk.lookupPort(client, "GoPort")
12 goport.go()
13 fwk.destroyInstance(server, 0.0)
14 fwk.destroyInstance(client, 0.0)
```

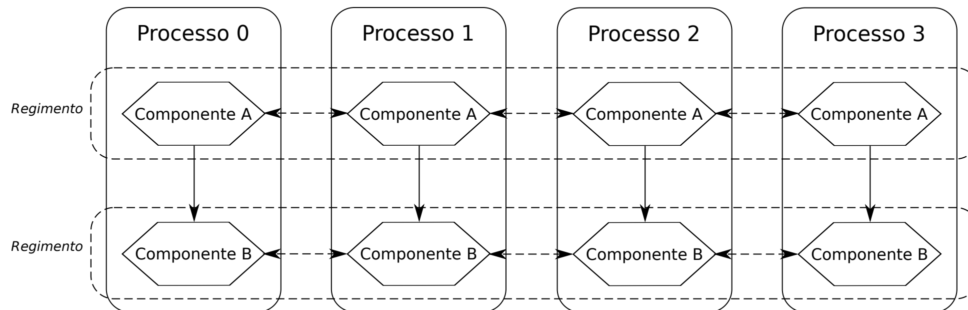
3.1.2 O Padrão de Projeto SCMD

Considerando os modelos de programação apresentados na Seção 2.2, o padrão de projeto mais comum para aplicações desenvolvidas usando MPI é o *Single Program Multiple Data* . Nesse padrão, vários processos são criados em nós de processamento utilizando como código um mesmo programa. Cabe ao desenvolvedor desse programa controlar o fluxo de cada processo através de informações sobre topologia e identificação, fazendo uso das diretivas de troca de mensagens para sincronização e compartilhamento de dados.

O modelo CCA adota um paradigma de desenvolvimento análogo chamado SCMD (*Single Component Multiple Data*) . Da forma semelhante ao SPMD, um programa idêntico é executado em cada processo participante em uma plataforma paralela e os dados são particionados entre os processos. Entretanto, para suportar componentes, a arquitetura CCA exige que cada um deles seja instanciado e configurado de forma idêntica em cada processo. Um conjunto de componentes instanciados dessa maneira é chamado de *regi-*

mento (do inglês *cohort*). Seja qual for o processo da aplicação, a arquitetura da conexão das portas entre as instâncias locais de um regimento é a mesma.

Figura 3.2: Diagrama do padrão de projeto SCMD.



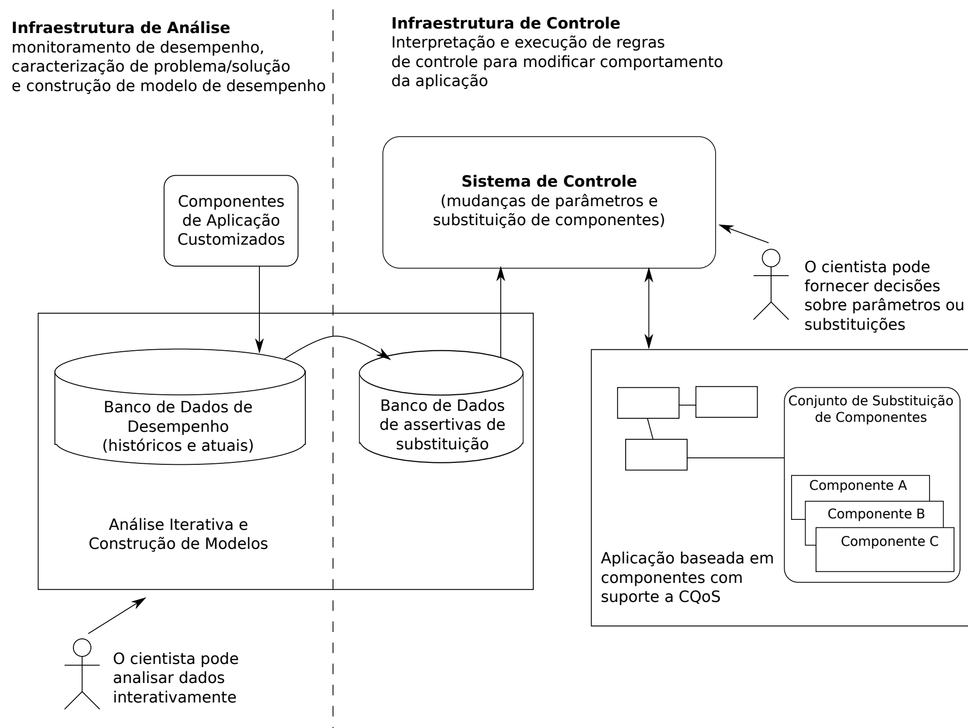
Fonte: (ARMSTRONG, 2006).

Uma representação do esquema SCMD está na Figura 3.2. Cada instância de componente em processo interage (linha horizontal tracejada) com outras instâncias em processos diferentes através de um paradigma de troca de mensagens. Dentro de um mesmo processo, as setas verticais representam invocações entre portas *uses* e *provides*. O paradigma SCMD, empregado com um conjunto uniforme de conexões entre as portas dos componentes, produz um programa SPMD para a aplicação geral. Portanto, os componentes em um mesmo *cohort* podem diferenciar suas instâncias para fazer o que for necessário para o algoritmo implementado. Qualquer invocação de método feita em uma interface deve ser realizada localmente em cada processo, obedecendo a mesma sequência lógica. A garantia desse comportamento fica a cargo do implementador do *framework*.

3.1.3 Reconfiguração Dinâmica de Componentes

Utilizando como base a CCA, os autores McInnes *et al* desenvolveram conceitos necessários para permitir controle adaptativo em tempo de execução de componentes (MCINNES, 2006b). Nesse trabalho, o objetivo é garantir a Qualidade de Serviço Computacional (*Computational Quality of Service* - CQoS) , definida como a capacidade que um sistema apresenta ao resolver um problema científico com as melhores ferramentas disponíveis de *software* ou *hardware*. A chave para um sistema habilitar a CQoS é a *adaptação de método*: durante pontos de decisão de uma computação em execução, a técnica ou módulo mais adequado é selecionada de forma dinâmica baseando-se no estado atual do problema. As diversas técnicas viáveis para determinada tarefa ou algoritmo científico podem ser implementadas como componentes, dessa forma permitindo a adaptação de método através da reconfiguração dinâmica do código da aplicação. A adaptação pode ser realizada através da atualização de parâmetros fornecidos por um componente ou pela substituição do mesmo.

Figura 3.3: Visão Geral de uma Infraestrutura com Suporte a CQoS.



Fonte: (MCINNES, 2006b).

A Figura 3.3 ilustra como uma infraestrutura para componentes de Computação de Alto Desempenho pode analisar, selecionar e parametrizar componentes. A *infraestrutura de análise* combina informações de desempenho e modelos criados a partir de históricos de execuções e informações das execuções correntes. Esses modelos podem servir de entrada para análise interativa, análise estatística e aprendizado de máquina. A *infraestrutura de controle* consiste de mecanismos capazes de efetuar a reconfiguração dinâmica dos componentes a partir das informações fornecidas pela análise. Essas duas infraestruturas interagem entre si através de assertivas armazenadas em um banco de dados compartilhado. O conteúdo de uma assertiva indica qual o comportamento esperado caso um parâmetro ou componente seja substituído. A análise constrói as assertivas com base nos modelos de desempenho, enquanto o controle as seleciona de acordo com o estado do sistema.

Para habilitar componentes para um ambiente com suporte a CQoS, não é indicado a exigência de mudanças extensas no código fonte, pois tais alterações dificultam o reuso de código legado. Entretanto, para permitir a reconfiguração, é necessário que o componente exponha na interface de suas portas variáveis *sensores* e métodos *atuadores*. As variáveis sensores representam qualquer informação capaz de caracterizar o desempenho atual do componente (por exemplo, o número de interações em um laço). Métodos atuadores dão acesso a parâmetros que podem alterar o comportamento de um componente

(por exemplo, a alteração de uma estratégia de distribuição de dados). O ideal é que a adição dos conceitos acima não impeçam um componente de também executar em uma infraestrutura sem suporte a CQoS.

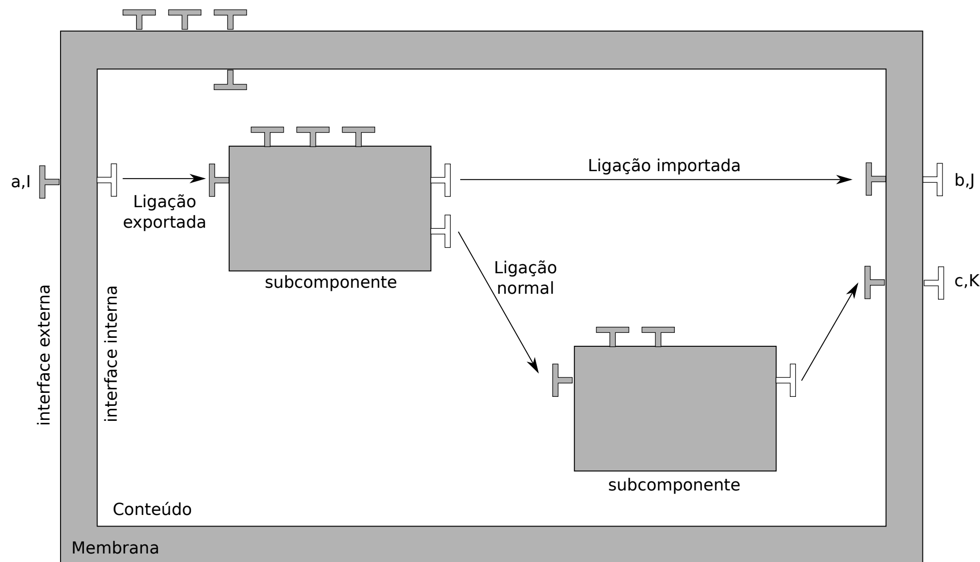
Para a infraestrutura de análise, a criação de modelos recebe como entrada medidas de desempenho, além de métricas específicas da aplicação. Para captura de dados, os autores propõe a utilização do sistema TAU (SHENDE; MALONY, 2006), porém com extensões para as métricas específicas. Quanto mais especializada for a análise de desempenho, melhores serão os resultados da reconfiguração. Um banco de dados armazena as informações de execuções anteriores de cada componente. O objetivo dos modelos criados é traduzir informações sobre as propriedades computacionais de um componente em assertivas preditivas de CQoS. Duas abordagens são consideradas para criação de modelos. A *abordagem analítica* define métricas de desempenho a partir da análise teórica da complexidade e função de isoeficiência (ver Seção 2.4). Já na *abordagem empírica*, o comportamento externo do componente é continuamente monitorado e comparado com o histórico dos outros componentes candidatos para a mesma funcionalidade.

A infraestrutura de controle utiliza os dados da análise para definir *proxies* para as portas *provides* de cada componente considerado para uma funcionalidade. Em outras palavras, um *proxy* representa um conjunto de componentes. Durante a execução, os *proxies* coletam informações e alimentam o modelo de desempenho. Com as assertivas definidas pela Análise, cada *proxy* decide qual componente de fato irá prover a funcionalidade desejada. É importante notar que mesmo sem esses artefatos intermediários, os componentes ainda poderiam ser orquestrados para uma aplicação com o mesmo resultado final, porém sem a reconfiguração dinâmica.

3.2 GCM

GCM (*Grid Component Model*) é um modelo de componentes hierárquico projetado para suportar aplicações autonômicas em plataformas distribuídas altamente dinâmicas e heterogêneas (BAUDE, 2009). Seu desenvolvimento foi motivado pelo advento das grades computacionais (ver Seção 2.1.2) como plataforma para Computação de Alto Desempenho. O GCM é uma extensão do modelo Fractal (BRUNETON, 2006) e dele herda a ênfase em composição hierárquica, capacidades de introspecção e reconfiguração de componentes. O Fractal define um modelo de componentes extensível que reforça a separação de interesses e a separação entre interface e implementação.

Figura 3.4: Composição de Componentes no Fractal.



Fonte: Adaptado de (BRUNETON, 2006).

3.2.1 Composição Hierárquica

Na Figura 3.4, temos um exemplo de composição hierárquica de componentes. O *conteúdo* é uma entidade abstrata controlada pelo *controlador*. O conteúdo de um componente é definido recursivamente por subcomponentes e ligações. O controlador (ou membrana) é a entidade que define a lógica de controle associada a um componente particular. Uma *interface servidor* é uma interface de componente que recebe invocações (item (a, I)). Uma *interface cliente* é uma interface que emite invocações (itens (b, J) e (c, K)). Essas interfaces correspondem às portas *provides/uses* do modelo CCA. O Fractal adota mais uma qualificação de interfaces, com o objetivo de enfatizar a separação de interesses. Uma *interface funcional* representa uma funcionalidade fornecida pelo componente para o propósito geral da aplicação, enquanto uma *interface de controle* é uma interface que permite controlar aspectos não funcionais tais como introspecção e reconfiguração. Na Figura 3.4, é apresentado apenas um nível de composição, mas os próprios subcomponentes podem apresentar interfaces externas que são membranas para uma composição mais interna. Essa flexibilidade permite que o *Fractal*, e por extensão o GCM, não determine um nível de granularidade restrito para decomposição de funcionalidades em componentes.

3.2.2 Comunicação Coletiva

Além das ligações primitivas para composição e invocação ponto a ponto do Fractal, o GCM dá suporte a interações coletivas para suportar comunicação no padrão M para N. Através das interfaces de *multicast* e *gathercast* operações coletivas de comunicação

podem ser implementadas sem a necessidade de componentes intermediários, o que tem impacto positivo no desempenho.

Uma interface *multicast* transforma uma invocação única em uma lista de invocações para componentes distintos. Considerando o aspecto cliente ou servidor de cada interface, uma interface servidor *multicast* transforma cada invocação única em um conjunto de invocações que são encaminhadas para um componente primitivo que implementa a interface ou para interfaces conectadas de componentes internos, no caso da composição hierárquica. Já uma interface cliente *multicast* transforma cada invocação única originária de um componente primitivo ou de um componente interno em um conjunto de invocações para interfaces servidor de componentes externos.

Uma interface *gathercast* transforma um conjunto de invocações em uma única invocação. Uma interface cliente *gathercast* transforma um conjunto de invocações vindo de interfaces cliente de componentes internos ou do próprio componente primitivo em uma única invocação. No caso da interface servidor *gathercast*, um conjunto de invocações originadas em interfaces servidor de componentes externos são transformadas em uma única invocação para uma interface servidor de um componente interno ou para a implementação do componente primitivo. Um dos benefícios advindos do uso da interface *gathercast* é a possibilidade de sincronizar um conjunto de invocações concorrentes em direção a um mesmo destino.

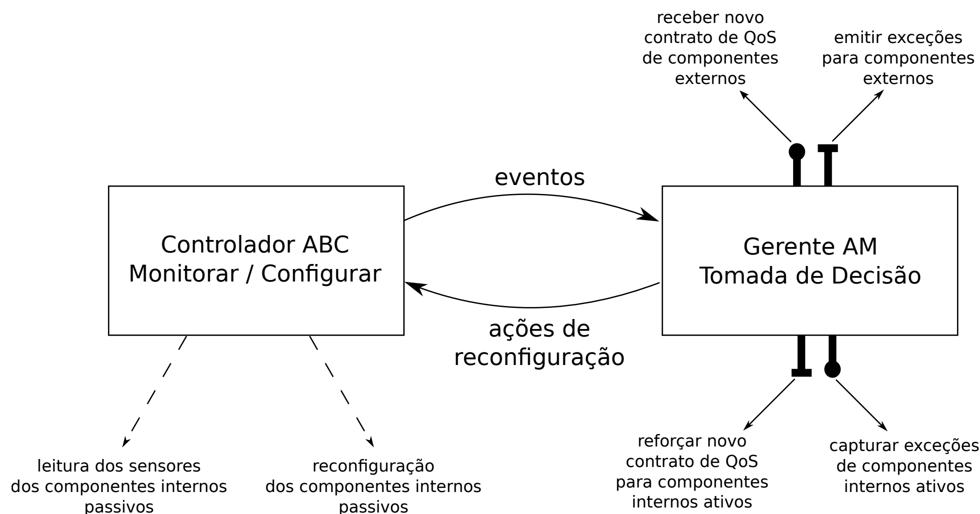
O problema definido pela comunicação entre dois programas paralelos, cada um formando por M e N entidades respectivamente, pode ser resolvido no GCM através da ligação de um componente paralelo com uma interface interna servidor *gathercast* com um componente com uma interface interna cliente *multicast*. Desta forma, o GCM apresenta a flexibilidade necessária para suportar uma diversa gama de aplicações de Computação de Alto Desempenho.

3.2.3 Gerência Autonômica e Adaptação no GCM

O GCM faz uso das interfaces de controle para implementar as atividades não funcionais, como a adaptação em tempo de execução. Assim como a composição adota um paradigma hierárquico, o GCM se apoia na definição de componentes internos para implementar a gerência autonômica. Cada componente, seja qual for o nível da hierarquia, contém um *Gerente Autonômico* (*Autonomic Manager* - AM), que interage com outros gerentes em outros componentes através de suas interfaces de controle. O gerente implementa um ciclo de decisão autonômico através de um programa simples baseado em regras reativas. Para atingir esse objetivo, o gerente interage com as membranas dos componentes para monitoramento de eventos e execução de ações de reconfiguração. Devido

a importância das interfaces de controle não funcionais na membrana, o conjunto delas recebe o nome de *Controlador de Comportamento Autonômico* (*Autonomic Behaviour Controller* - ABC). As interfaces não funcionais de um controlador são todas do tipo servidor, sendo que podem ser acessadas diretamente por um gerente ou por um componente externo. Um componente que possui apenas o elemento controlador ABC é dito *passivo*, enquanto componente que apresente tanto o ABC quanto o gerente AM é chamado de *ativo*.

Figura 3.5: Interação entre gerentes e controladores.



Fonte: Adaptado de (ALDINUCCI, 2008).

A Figura 3.5 apresenta uma visão da interação entre o gerente e o controlador em um nível da hierarquia de composição. O gerente executa sua lógica local, mas interage com os componentes externos, tanto ao receber novos requisitos de QoS, quanto para enviar informar sobre exceções no cumprimento desses requisitos. Ele também encaminha para gerentes de componentes internos o novo contrato de QoS. Além de interagir com os outros elementos ativos internos, o gerente atua no controlador local emitindo ações de reconfiguração, que por sua vez são repassadas para componentes internos passivos. Esses componentes também são monitorados pelo controlador. Em uma aplicação complexa, com vários níveis na hierarquia, exigir que os desenvolvedores definam cada implementação dos gerentes e dos controladores é um cenário indesejado, pois além da funcionalidade do componente, o programador precisa entender os princípios da gerência autonômica.

3.2.3.1 Esqueletos Comportamentais

Esqueletos comportamentais (*Behavioural Skeletons*) têm como objetivo abstrair paradigmas configuráveis de uma aplicação GCM, cada um deles especializado na resolução de um ou mais metas gerenciais tais como configuração, otimização e proteção (ALDINUCCI,

2008). Assim como esqueletos algorítmicos, eles representam padrões de computação paralela (que são representadas no GCM como grafos de componentes), mas também usam a semântica inerente da arquitetura representada para projetar esquemas de auto gerência corretos a partir de componentes paralelos. A primeira vantagem dessa abordagem é permitir que os *usuários finais* de uma aplicação GCM possam reaproveitar funcionalidade de componentes já existentes, fornecendo apenas parâmetros de QoS de alto nível para a lógica de decisão de cada gerente AM. Caso seja necessário, um *usuário especialista* em computação autonômica pode reescrever a lógica de decisão dos gerentes para tratar de novos requisitos. Já o *projetista de esqueletos*, com profundo conhecimento de padrões de paralelismo e técnicas autonômicas, pode estudar uma classe de problemas e definir um novo esqueleto comportamental, definindo novas interfaces controladoras ABC para cada classe de componente que faz parte da hierarquia arquitetural.

Dada a composição hierárquica no GCM, podemos considerar um esqueleto comportamental como um componente de alta ordem que ao mesmo tempo que expõe a descrição do seu comportamento funcional, estabelece um esquema parametrizado para orquestração de seus comportantes internos. Seu comportamento externo pode apresentar restrições que são originadas pelos seus componentes internos, além de já possuir um conjunto de estratégias pré-definidas para atingir determinada meta de auto gerência.

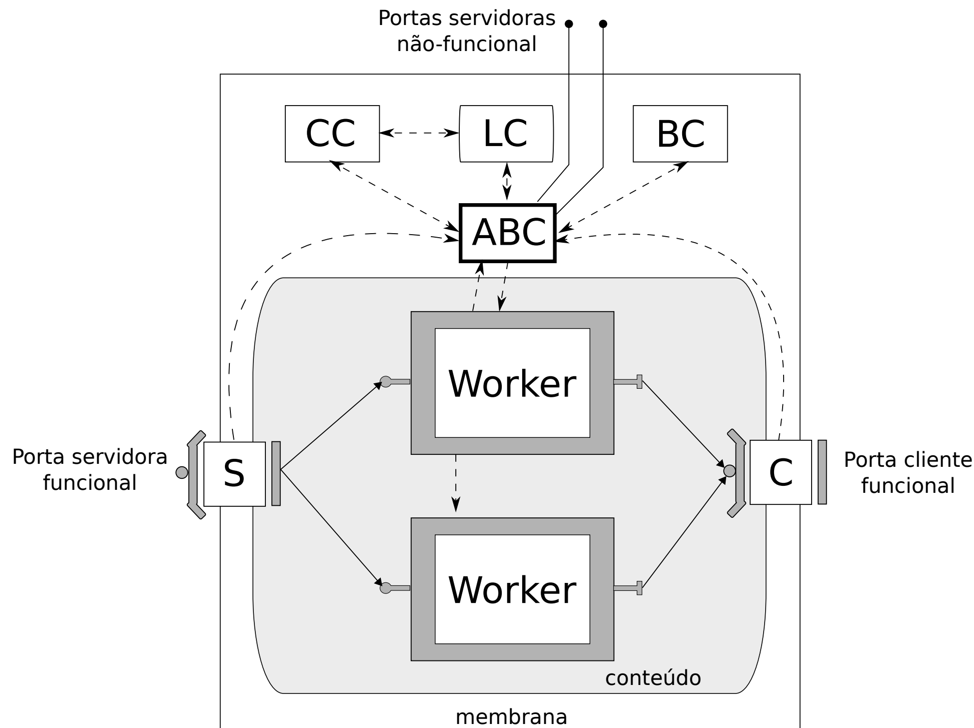
3.2.3.2 Arquitetura de Componentes Autonômicos

Para exemplificar a utilização de esqueletos comportamentais, considerem o padrão paralelo de replicação funcional. Neste padrão, o esqueleto apresenta uma interface servidora *multicast* que despacha requisições para serem processadas em tarefas executadas por componentes internos ditos trabalhadores (*Worker*). Os resultados das tarefas são consolidados por uma interface cliente *gathercast*, que emite um resultado único para os componentes externos ao esqueleto.

Na Figura 3.6, um componente autonômico passivo apresenta o controlador ABC interagindo com o controlador de ciclo de vida (*Lifecycle Controller* - LC), o controlador de ligação (*Binding Controller* - BC) e com o controlador de conteúdo (*Content Controller* - CC). Para o padrão de paralelismo do exemplo, o componente passivo permite que gerentes AM externos controlem o número de *Workers*. Para habilitar essa decisão, o componente deve fornecer informações sobre o tráfego de informações nas interfaces e a taxa de produção de cada componente interno.

A principal diferença entre o componente passivo e o ativo é a presença do gerente interno AM. Na Figura 3.7, o gerente interno com o controlador ABC, com os componentes trabalhadores e com as interfaces de entrada e saída. Ao contrário do componente passivo, ele não recebe do ambiente externo ações que devem ser executadas diretamente. O

Figura 3.6: Componente Autonômico Passivo.



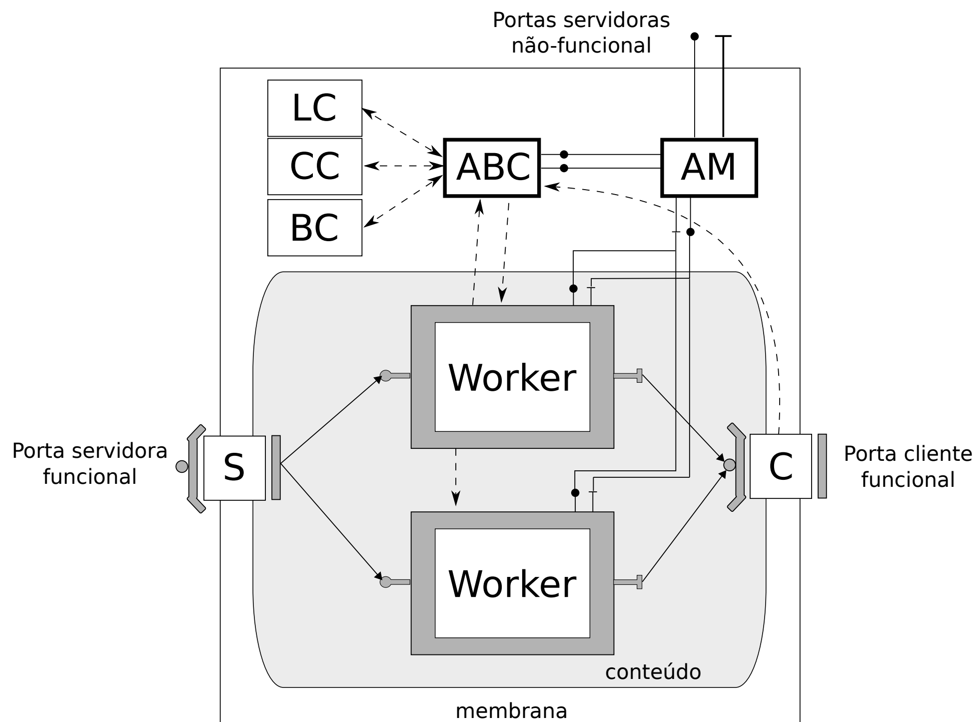
Fonte: Adaptado de (ALDINUCCI, 2008).

gerente AM recebe um contrato de QoS, que pode ser definido como um par $< V, E >$, no qual V é um conjunto de variáveis representando as métricas que o AM pode avaliar (através de leituras no ABC) e E é uma expressão matemática sobre essas variáveis, que pode incluir os operadores *min* ou *max* sobre um domínio finito. Durante a execução, o gerente continuamente avalia a expressão E de acordo com os valores medidos pelo controlador. No caso de uma das variáveis apresentar valor que invalide a expressão E, o AM deve executar um plano de reconfiguração. Por exemplo, no caso do padrão de replicação funcional, um contrato de QoS que determine uma taxa mínima de produção de valores na saída pode levar o gerente a instanciar um plano que cria uma quantidade maior de componentes trabalhadores. A dificuldade na definição de planos de reconfiguração pode ser minimizada através da composição de outros esqueletos comportamentais.

3.3 Conclusões

Neste capítulo, apresentamos modelos baseados em componentes para o desenvolvimento de aplicações de Computação de Alto Desempenho e suas respectivas soluções para reconfiguração. Um ponto importante a ressaltar é que o foco de ambas soluções é a reconfiguração dos elementos de *software* da arquitetura. Apesar de implicitamente a reconfiguração de um componente de *software* acarretar uma mudança na plataforma,

Figura 3.7: Componente Autonômico Ativo.



Fonte: Adaptado de (ALDINUCCI, 2008).

nos trabalhos estudados não é possível isolar e rastrear as alterações nos recursos. Isto ocorre em parte porque a infraestrutura computacional não é representada da mesma forma que os elementos de *software*, através da abstração pelo uso de componentes. Essa opção foi natural na época de desenvolvimento dos modelos (início dos anos 2000), pois as infraestruturas disponíveis (*clusters* e grades computacionais) apesar de apresentarem contextos de execução dinâmicos, não possuíam interfaces de programação maduras e o conjunto de recursos sob controle era limitado.

Com o advento das nuvens computacionais, a barreira de programabilidade dos recursos computacionais foi flexibilizada. O conceito de elasticidade permite configurar a infraestrutura com um alto nível de granularidade de opções, sua aplicação em nuvens computacionais abre a oportunidade de definir uma arquitetura de componentes para Computação de Alto Desempenho com maior nível de flexibilidade na reconfiguração do ambiente. No Capítulo 4, nos aprofundamos no estudo desse conceito e sua importância para a Computação de Alto Desempenho.

4. Elasticidade em Nuvens Computacionais

A elasticidade é um característica das nuvens computacionais que permite a gestão dinâmica dos recursos. Uma aplicação pode, através da execução elástica, utilizar conjuntos diferentes de recursos em momentos distintos do seu ciclo de vida. Essa característica facilita a alocação conjunta de aplicações em uma mesma infraestrutura. Desde que o pico de utilização das aplicações não coincida, os usuários não percebem degradação no desempenho, sendo mantida a ilusão de que toda a infraestrutura é de uso exclusivo. Para as aplicações comerciais, que estão sujeitas a grande variação de demanda, essa característica das nuvens levou ao desenvolvimento de inúmeras técnicas de controle automático de escalabilidade.

As aplicações de Computação de Alto Desempenho, apesar de terem requisitos diferentes dos códigos comerciais, também podem tirar proveito da elasticidade. Nesse caso, não há variação de demanda durante o ciclo de vida da computação, mas mesmo uma entrada fixa na submissão de uma tarefa pode apresentar requisitos de processamento distintos no decorrer da execução. Há também a situação em que, devido a várias aplicações dividirem recursos de uma mesma infraestrutura, a execução de uma acabe influenciando o desempenho das outras, levando a necessidade da reorganização dos recursos.

De acordo com o nosso objetivo de habilitar a execução eficiente de aplicações científicas nas infraestruturas modernas, neste capítulo realizamos um levantamento do estado da arte em controle de escalabilidade na nuvem. Em seguida, fazemos um paralelo entre elasticidade na computação comercial e sua aplicação na Computação de Alto Desempenho. Apesar das características diferentes das aplicações, a elasticidade no sentido geral oferece direções que podem ser aproveitadas no contexto de Computação de Alto Desempenho. Ao final, apresentamos o estado da arte para elasticidade em programas paralelos.

O restante do capítulo é organizado da seguinte maneira. Partimos de uma visão geral de elasticidade em nuvens computacionais (Seção 4.1). Em seguida, apresentamos nossa definição de elasticidade para Computação de Alto Desempenho (Seção 4.2). Por fim, discutimos as soluções existentes que têm objetivo semelhante ao deste trabalho (Seção 4.2.3).

4.1 Elasticidade para Aplicações Comerciais

Apresentamos nesta seção um resumo das técnicas de elasticidade utilizadas em aplicações comerciais. Apesar desta Tese tratar de Computação de Alto Desempenho, há conceitos e técnicas da computação comercial que podem ser reaproveitados (LORIDO-BOTRAN; MIGUEL-ALONSO; LOZANO, 2014).

4.1.1 Conceitos Gerais

Dada uma nuvem IaaS e uma aplicação para ser implantada, podemos identificar três atores: *provedor, cliente* e *usuário*. O provedor é o administrador da nuvem IaaS que fornece as máquinas virtuais. O cliente é o desenvolvedor da aplicação que deseja hospedá-la na nuvem. Já o usuário é quem faz uso da aplicação, gerando a carga de trabalho a qual ela será submetida. Dada uma aplicação qualquer, a elasticidade da nuvem apresenta desafios. Ao mesmo tempo que permite a aquisição de mais recursos pela aplicação de maneira dinâmica, decidir qual a quantidade ideal de recursos não é uma tarefa fácil. Para todos os envolvidos, o desejável é um sistema que ajuste automaticamente a quantidade de recursos de acordo com a carga de trabalho submetida à aplicação. Isso resultaria em um sistema com a característica de auto escalabilidade.

A elasticidade pode ser *horizontal* ou *vertical*. Na elasticidade horizontal, os recursos são instâncias de máquinas virtuais, que podem ser adicionadas (*scale-out*) ou removidas (*scale-in*) da aplicação. Elasticidade vertical trata da alteração dos recursos que estão designados para uma máquina virtual, por exemplo, aumentando (*scale-up*) ou diminuindo (*scale-down*) núcleos virtuais ou a memória de uma máquina virtual. Apesar de alguns *hypervisors* como o XEN permitirem a reconfiguração dinâmica da memória e quantidade de núcleos em máquinas virtuais, nem todos os provedores de nuvens públicas fornecem essa opção.

O controle da elasticidade de uma aplicação precisa considerar o *custo financeiro* da execução na nuvem. O modelo atual de cobrança se baseia no uso por unidade de tempo (minutos ou horas) e no perfil da máquina virtual instanciada. Esse perfil é a descrição do *hardware* utilizado.

4.1.2 Aplicações Comerciais Elásticas

Para uma aplicação *web*, elasticidade é a capacidade de se adaptar através de ações horizontais ou verticais de acordo com a carga de trabalho variável. Aplicações desse tipo em geral são estruturadas em dois níveis. No primeiro nível, estão os balanceadores de

carga, enquanto, no segundo nível, estão os servidores que armazenam as aplicações e o conteúdo. Uma requisição do usuário é inicialmente tratada pelo sistema de balanceamento de carga que decide qual servidor de conteúdo atenderá o pedido.

A elasticidade é controlada pelo sistema de balanceamento de carga. Quando detecta que os servidores de conteúdo estão sobrecarregados, o balanceamento de carga instancia novas máquinas virtuais no segundo nível. Na situação inversa, quando os servidores de conteúdo estão ociosos, máquinas virtuais são desalocadas. Um servidor do primeiro nível também pode decidir clonar sua própria instância caso perceba que está sobrecarregado ou desligar-se caso esteja ocioso.

Para tal esquema funcionar, o balanceamento precisa ter informações atualizadas sobre o estado das máquinas virtuais e dos custos de utilização da nuvem. Não incluímos na nossa discussão detalhes sobre replicação e compartilhamento de dados, pois foge do escopo da Tese. O leitor interessado pode encontrar mais detalhes em (SOUSA; MACHADO, 2012).

4.1.3 Gerência de Elasticidade

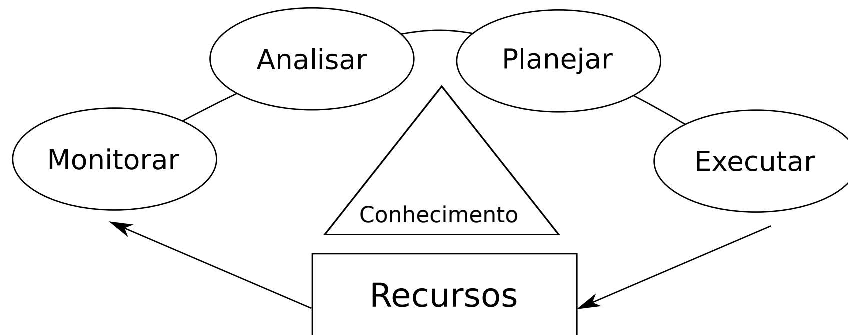
Considerando o universo das aplicações comerciais elásticas, nesta seção discutimos como os desenvolvedores podem controlar a reconfiguração do código e infraestrutura.

4.1.3.1 O Laço MAPE

O objetivo da elasticidade é adaptar dinamicamente os recursos designados para as aplicações, dependendo da carga de trabalho. Um sistema de gerência de elasticidade deve ser capaz de encontrar um equilíbrio entre a satisfação dos clientes e usuários e o custo do uso da nuvem. Para atingir esse objetivo, o ambiente precisa lidar com três situações. O primeiro problema envolve o caso onde a aplicação está sem os recursos necessários em um dado momento (*under provisioning*) e requisita mais máquinas virtuais ao provedor, o qual demora para atender à requisição, levando a uma queda da qualidade de serviço. O segundo problema ocorre quando, por um erro no projeto, a aplicação acaba requisitando mais recursos do que realmente precisa (*over provisioning*), causando prejuízo financeiro. O terceiro problema é uma combinação dos dois primeiros, quando as ações de elasticidade são tomadas em intervalos muito curtos, resultando em oscilação.

Uma saída para tratar os problemas é adotar o laço MAPE de sistemas autônomos (MAURER, 2011). Como observamos na Figura 4.1, MAPE significa Monitorar, Analisar, Planejar e Executar . Essas fases compartilham informações entre si através de um repositório de conhecimento.

Figura 4.1: O laço MAPE.



Fonte: Adaptado de (KEPHART, 2007).

O *monitoramento* recupera informações do ambiente para calcular métricas de desempenho. A eficiência da adaptação elástica vai depender da qualidade das métricas, a regularidade da coleta de dados e a sobrecarga da obtenção de dados. Como exemplo de métricas, podemos citar o percentual de utilização da CPU por máquina virtual, uso total de memória, o número de requisições na fila, etc. A definição das métricas é feita pelo cliente da nuvem, sendo a norma variar de acordo com a aplicação.

A *análise* consiste no processamento dos dados do monitoramento para gerar informações sobre o estado futuro da aplicação. O controle de elasticidade mais simples pode evitar realizar qualquer previsão e somente reagir às mudanças imediatas na carga de trabalho. Nesse caso, é chamada de *elasticidade reativa*. Outra opção é utilizar técnicas sofisticadas para realizar previsões e configurar a aplicações para mudanças vindouras. Temos então a *elasticidade proativa*. Previsões corretas minimizam a sobrecarga de criação de máquinas virtuais. Quando o recurso se torna necessário, já encontra-se instanciado.

Uma vez que a análise fornece uma previsão dos próximos eventos, o *planejamento* permite definir qual o conjunto de ações é necessário para manter a qualidade de serviço. Um exemplo de ação é adicionar ou remover uma máquina virtual em um dado momento. Além da informação dinâmica fornecida pela análise, o planejamento leva em conta dados fixos como a tabela de custo do provedor de nuvem.

A fase final do laço, a *execução*, é aquela mais simples conceitualmente. É implementada ao executar as ações fornecidas pelo planejamento através da API que o provedor de nuvem IaaS fornece.

4.1.3.2 Técnicas de Elasticidade

As técnicas para controlar elasticidade de aplicações e implementar as fases do laço de controle podem ser classificadas de acordo com o modelo teórico que dá suporte e formaliza as fases de análise e planejamento. Os cinco principais arcabouços teóricos são:

- Regras baseadas em limiares;
- Aprendizagem por reforço;
- Teoria das Filas;
- Teoria do Controle;
- Análise de Séries Temporais.

Amazon EC2 e Microsoft Azure, provedores de nuvem públicas IaaS, oferecem apenas elasticidade usando regras baseadas em limiares com ênfase na fase de planejamento. As decisões de adaptação são tomadas a partir de métricas de desempenho e limiares pré-definidos, caracterizando uma técnica reativa. Uma limitação dessa técnica é a dificuldade em lidar com mudanças repentinas. Cada regra tem duas partes: a condição e a ação a ser executada quando a condição é alcançada. A condição é construída a partir de cláusulas que utilizam métricas de desempenho. Um exemplo de regra seria: *adicione duas instâncias pequenas quando a média de utilização da CPU está acima de 70% durante mais do que 5 minutos.*

Na aprendizagem por reforço (SUTTON; BARTO, 1998), o sistema de elasticidade aprende, a partir da experiência, a melhor adaptação a ser feita, de acordo com o estado atual da aplicação, que é definido pela carga de trabalho, desempenho e outras variáveis. Após a execução de uma ação, o sistema (agente) recebe uma resposta (recompensa) que informa o efeito da ação. Com o decorrer do tempo, o agente aprende quais são as ações positivas e quais são as negativas e passa a atuar em busca de garantir a qualidade de serviço. É uma técnica reativa, sendo que a coleta das respostas de ações anteriores é a fase de análise enquanto a decisão das próxima ações de acordo com o aprendizado corresponde a fase de planejamento.

A Teoria das Filas (KLEINROCK, 1976) é usada em várias áreas da computação. Entretanto, para o cenário de elasticidade, apresenta pouca flexibilidade, sendo usada na fase de análise através do estudo da fila de submissões e do tempo médio de resposta para cada requisição. Quando observa mudanças nesses parâmetros, é possível reagir provendo adaptação. Trata-se de uma técnica reativa.

A Teoria do Controle (GARCIA; PRETT; MORARI, 1989) define que um controlador deve ser definido de acordo com o modelo da aplicação que irá tratar das fases de análise e planejamento. O controlador tem como objetivo manter uma determinada variável de estado do sistema dentro de uma faixa de valores. Controladores de laço aberto utilizam apenas o estado das variáveis para tomar decisões, não são realimentados com o resultado das ações. Por sua vez, controladores com *feedback* são semelhantes a aprendizagem por reforço, já que armazenam as respostas das ações para guiar decisões futuras. Finalmente,

controladores *feed forward* tentam prever cenários futuros de acordo com os erros das decisões passadas. Portanto, a Teoria de Controle torna possível tanto um cenário reativo (*feedback*) quando proativo (*feedback + feed forward*).

Análise de séries temporais (HAMILTON, 1994) são métodos capazes de detectar padrões e prever valores futuros a partir do estudo de uma sequência de pontos, medidos de maneira sucessiva em intervalos de tempo uniformes. Trata-se de uma técnica usada na fase de análise e sua acurácia depende da escolha adequada da janela histórica e do intervalo de previsão. Como fornece uma previsão, a análise de séries é uma técnica proativa.

Além das técnicas citadas, trabalhos recentes apresentam outras metodologias semelhantes baseadas em análise estatística (BENZ; BOHNERT, 2015). Por exemplo, Coutinho *et al* utilizam o conceito de computação autonômica para definir uma arquitetura que serve como base para implementação de soluções elásticas com suporte às técnicas estudadas (COUTINHO; GOMES; SOUZA, 2015). Um ponto em comum a todas as alternativas apresentadas é que consideram a aplicação se adaptando a variação na demanda, alocando ou liberando recursos para manutenção de um acordo de qualidade de serviço. Em outras palavras, a variação do contexto que inicia a adaptação ocorre na carga de trabalho, não no conjunto de recursos disponíveis.

4.2 Elasticidade em Computação de Alto Desempenho

Nesta seção, apresentamos uma definição de elasticidade com o foco em Computação de Alto Desempenho, além de discutirmos o estado da arte de sua aplicação.

4.2.1 Definição de Elasticidade para CAD

A discussão apresentada na seção anterior considera a definição tradicional de elasticidade. O *National Institute of Standards and Technology* (NIST) menciona, no seu documento sobre normas técnicas das nuvens computacionais (MELL; GRANCE, 2011), o seguinte sobre elasticidade:

> ***Rapid elasticity**: Capabilities can be rapidly and elastically provisioned, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.*

Essa definição está correta para as aplicações comerciais mencionadas na Seção 4.1.2. Podemos afirmar que a maioria das definições de elasticidade estão em sintonia com a

definição do NIST. Em (COUTINHO, 2015), os autores apresentam uma lista ampla de definições encontradas na literatura. Para o contexto desta Tese, podemos destacar três pontos conceituais que estão presentes na maior parte das definições encontradas na literatura:

1. Alocação ou liberação de recursos de maneira facilitada ou até mesmo automática;
2. Capacidade da aplicação se adaptar, ou seja, fazer uso eficiente dos novos recursos alocados ou executar em um conjunto menor de recursos em relação ao inicialmente alocado;
3. A demanda ou carga de trabalho guiando o processo de adaptação elástica.

Para o caso de aplicações de Computação de Alto Desempenho, os dois primeiros pontos continuam relevantes. Entretanto, o papel da demanda (terceiro ponto) precisa ser reconsiderado. No contexto inicial das nuvens (aplicações comerciais), os sistemas não utilizam todos os recursos de maneira uniforme durante seu ciclo de vida. Por exemplo, uma aplicação de comércio eletrônico com certeza utilizará mais recursos durante o período de final de ano do que no resto do ano. Portanto, para esse tipo de aplicação, nada mais natural que a demanda seja o principal motor da elasticidade.

Em aplicações CAD, geralmente a entrada é fornecida no início e toda a execução é necessária para fornecer a resposta para o problema, não havendo alta variação no nível de utilização dos recursos alocados, sendo comum a utilização uniforme e completa do que está disponível. O imperativo não é se adaptar a demanda, mas sim garantir melhor desempenho, apenas aproveitando novos recursos disponíveis após o início da execução se a melhoria no desempenho for comprovada. Com essa diferença em foco, o conceito de elasticidade para Computação de Alto Desempenho pode ser tido como:

> *Elasticidade em Computação de Alto Desempenho é a capacidade de um ambiente que garante a gerência adaptativa de recursos computacionais com o objetivo de aprimorar o desempenho de uma aplicação, considerando as limitações de escalabilidade dos algoritmos utilizados e os requisitos não funcionais do usuário.*

A gerência adaptativa garante a alocação e liberação de recursos. O desempenho é o conceito que direciona esse processo, mas restrições como custo da execução e consumo energético devem ser levadas em conta, dentre outros requisitos não funcionais. A escalabilidade de algoritmos diz respeito a isoeficiência (ver Seção 2.4). Adicionar recursos a uma aplicação nem sempre se traduz em melhora do desempenho. Para alguns padrões

de distribuição com baixo acoplamento, como o *bag of tasks*, essa restrição não é forte. Porém, para aplicações fortemente acopladas, a inclusão de instâncias pode desequilibrar o balanceamento entre computação e comunicação. Dessa forma, o processo de adaptação deve considerar a natureza do algoritmo.

A demanda desempenha papel importante para ambientes de Computação de Alto Desempenho sob o ponto de vista do mantenedor de infraestruturas. Considere um centro de pesquisa que mantém um supercomputador (*cluster*) acessado pelos cientistas através de uma conta em um gerenciador de filas. O mantenedor pode configurar o sistema para atuar junto com uma nuvem computacional, alocando máquinas virtuais que recebem tarefas da fila quando o *cluster* está sobrecarregado. Nesse caso, a aplicação que se adapta não é o código paralelo dos pesquisadores, mas sim o gerenciador de recursos, que controla a fila de submissão. A aplicação paralela alocada utiliza o mesmo conjunto de recursos durante toda execução.

Com a definição de elasticidade para aplicações de Computação de Alto Desempenho estabelecida, nas próximas seções discutimos trabalhos revelantes sobre esse tema.

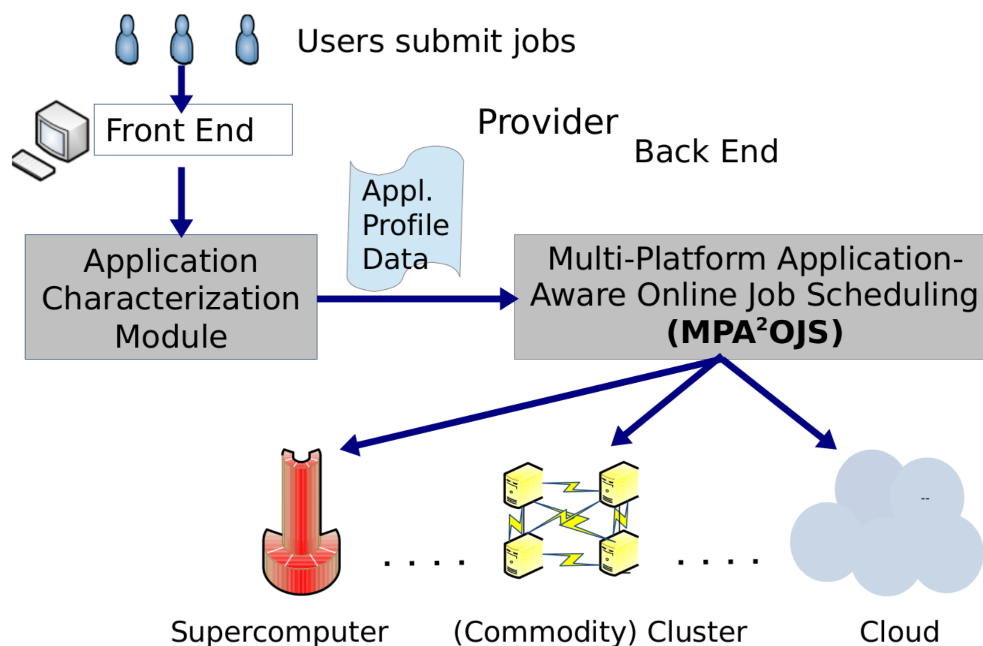
4.2.2 Gerência Elástica de Recursos

Considerado um supercomputador com uma fila e um gerenciador de recursos, máquinas virtuais de uma nuvem pública ou privada podem ser utilizadas para criar um *cluster* virtual e atender parte das tarefas da fila. A Figura 4.2 apresenta um exemplo de arquitetura que suporta o escalonamento elástico.

Os usuários submetem tarefas para o *Front End* do gerenciador de recursos, que é habilitado com componentes que auxiliam na decisão de qual plataforma deve ser escolhida para execução de cada submissão. O módulo *Application Characterization Module* analisa qual o tipo de aplicação a tarefa representa. Por exemplo, no caso de aplicações sensíveis à latência da rede de interconexão, as nuvens são descartadas como plataformas. O módulo *Application Profile Data* utiliza o histórico de execuções anteriores para enriquecer o processo de decisão. Com essas informações disponíveis, o gerenciador de recursos multiplataforma decide qual infraestrutura deve utilizar, seja supercomputadores, *clusters* de médio porte ou máquinas virtuais na nuvem.

Gupta (2014) (GUPTA, 2014) apresenta uma extensão da arquitetura que permite que o gerenciador de recursos adapte a execução da tarefa dentro de uma mesma plataforma. Por exemplo, caso um *cluster* seja escolhido para executar uma tarefa e inicialmente dois nós de processamento são alocados. Se durante a execução mais nós de processamentos forem liberados por outras tarefas, o gerenciador de recursos pode expandir a tarefa para ocupar os novos recursos ociosos. Para tal, a tarefa tem que ser código maleável (ver

Figura 4.2: Mapeamento de Aplicações CAD em várias plataformas.



Fonte: Adaptado de (GUPTA, 2014).

Seção 2.3.2).

Outros trabalhos adotam variação dessa abordagem (CABALLER, 2013) (MATEESCU; GENTZSCH; RIBBENS, 2011) (HUANG, 2013) (NIU, 2013). Apesar de oferecer vantagens ao usuário como redução do tempo de fila, o mesmo tem pouco controle sobre a execução nessas soluções. O administrador é o responsável pelo gerenciador de recursos e é quem define as regras, limitando a capacidade do desenvolvedor. Por exemplo, caso a tarefa seja uma aplicação que nunca foi executada no sistema, o gerenciador de recursos pode adotar decisões inadequadas.

4.2.3 Suporte à Elasticidade em Nível de Programação

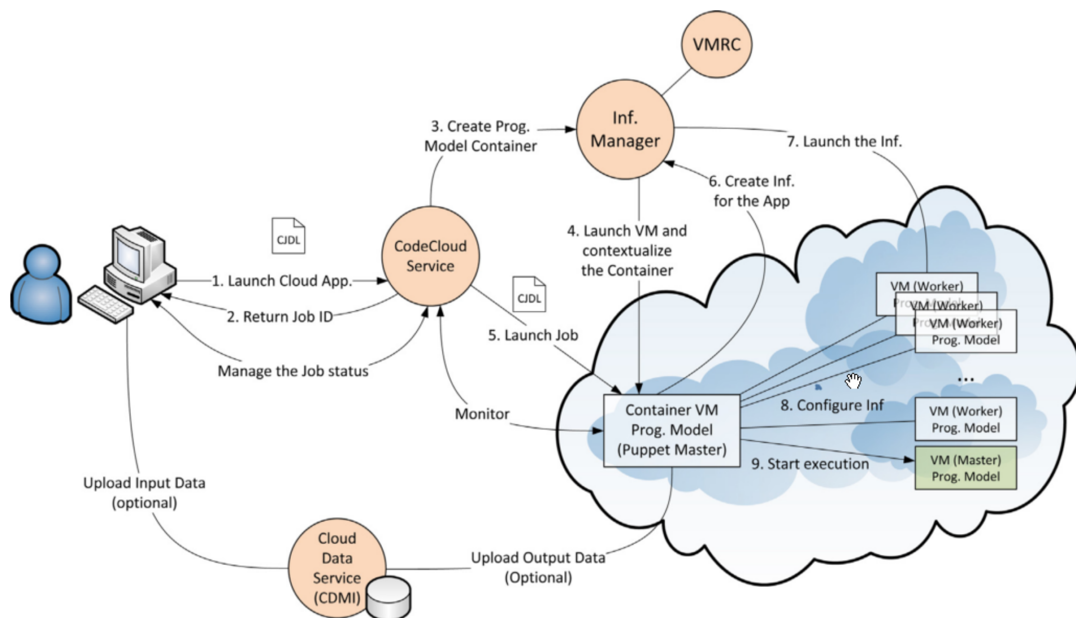
Nesta seção, discutimos trabalhos que permitem ao desenvolvedor controlar a elasticidade à partir da própria aplicação ou que exigem alterações arquiteturais para habilitar a reconfiguração.

4.2.3.1 *CodeCloud*

No primeiro nível da escala em termos de habilitar o desenvolvedor para controlar a elasticidade, temos a possibilidade do usuário adicionar na descrição da tarefa informações que ajudem ao gerenciador de recursos guiar a adaptação. (CABALLER, 2014) descrevem

como os modelos de programação MapReduce e mestre/escravo são aceitos pelo ambiente CodeCloud, além do MPI. Entretanto, para o ambiente, a aplicação é encapsulada em um componente tarefa, sendo o modelo de programação uma informação útil apenas na configuração das plataformas, guiando o carregamento de bibliotecas e detalhes da rede. Para configurar as tarefas, temos a linguagem específica de domínio CJDL (*Cloud Job Definition Language*). Essa linguagem permite ao usuário final especificar quais recursos necessita e qual o modelo de programação a ser utilizado. Além disso, ela suporta uma extensão à linguagem de *workflows* WINGS, para a definição de fluxos de execução.

Figura 4.3: Submissão de aplicação no CodeCloud.



Fonte: Adaptado de (CABALLER, 2014).

Na Figura 4.3, após o usuário definir sua computação através da CJDL (passo 1), um contêiner chamado EVC (*Enhanced Virtual Container*) é criado (passo 3), contendo as informações da CJDL e um modelo de máquina virtual para gerenciar a execução na nuvem de uma aplicação de acordo com o modelo de programação definido na CJDL. Esse contêiner é instanciado na nuvem (passo 4) e trata então de criar as máquinas virtuais (*workers*) necessárias (passo 8), bem como copiar os dados para iniciar a execução (passo 9).

CodeCloud permite que regras de elasticidade sejam definidas pelo usuário (técnica de regras baseadas em limiares). Essas regras atuam sobre informações disponíveis pelo sistema de monitoramento Ganglia. A plataforma também permite que ações sejam tomadas de acordo com informações específicas da aplicação, tais como o valor de uma variável. Sejam informações do Ganglia ou da aplicação, as métricas para elasticidade são armazenadas em um catálogo, sendo que a decisão nunca é tomada pelo último valor de

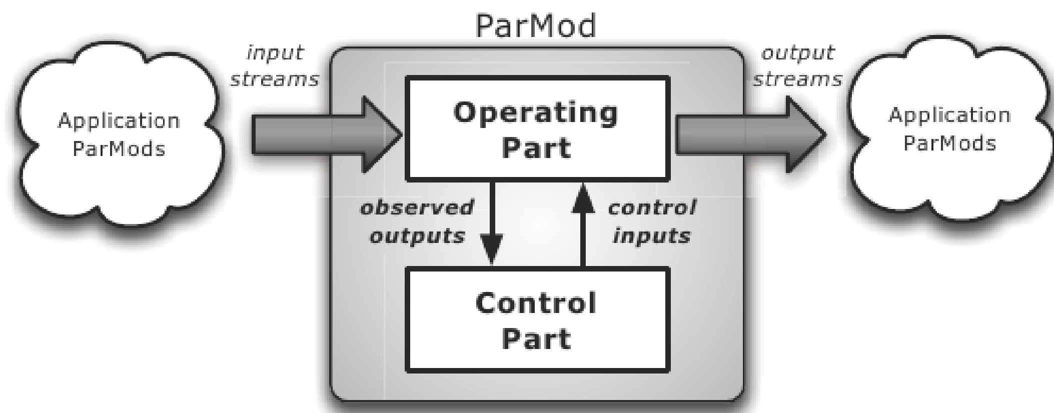
cada métrica, mas sim pela análise estatística de todas elas desde o início da execução. O responsável pela ação de elasticidade é o gerenciador de recursos, seguindo as orientações contidas no CJDL. Dessa forma, o controle da elasticidade por parte do desenvolvedor é limitado. Apesar de utilizar um programa MPI no estudo de caso, esse código é organizado no paradigma mestre/escravo, sem comunicação entre os processos trabalhadores. A elasticidade atua então apenas no nível de criação de máquinas virtuais para os processos trabalhadores, não sendo evidenciada a eficácia da solução para aplicações altamente acopladas.

4.2.3.2 ParMod

Enquanto o ambiente CodeCloud se assemelha a um gerenciador de recursos elástico, ParMod permite modelar aplicações em um conjunto de componentes interligados formando um grafo de dependência (MENCAGLI; VANNESCHI; VESPA, 2013). Cada componente é mapeado de maneira estática a uma plataforma de nuvem. Através da técnica de Teoria de Controle, o componente aloca os recursos necessários na plataforma para garantir a execução equilibrada do grafo. O objetivo é permitir a adaptação em tempo de execução para cada componente. O conceito de ParMod (*Adaptive Parallel Module*) é definido como um componente responsável pela execução de uma computação paralela que possui uma estratégia de adaptação definida. Esse componente é dividido em duas partes: *parte operacional* (*Operating Part*), que executa a computação em si, e *parte de controle Control Part*, responsável pela implementação da estratégia de adaptação. Dessa forma, a parte de controle é responsável por monitorar o progresso da computação na parte operacional. A adaptação realizada pode envolver alterações no grau de paralelismo (criação de novas máquinas virtuais), ou funcional, tratando da adaptação do código em ambientes heterogêneos. O código da parte operacional pode ser escrito em qualquer modelo de programação que a plataforma suporte. Uma aplicação distribuída é composta por vários componentes interconectados, sendo que cada unidade pode executar em uma plataforma (*cluster* real ou virtual) diferente.

Na Figura 4.4, temos a representação de um componente. No componente que representa a parte de controle há um laço de monitoramento similar ao laço MAPE. Nesse laço, há uma modelagem para o comportamento da parte operacional. É utilizado o conceito de estados baseados nos valores de variáveis internas, que são adaptados de acordo com medidas de perturbação no ambiente que resultam em entradas pela parte de controle. Apesar da modelagem ser generalista, no estudo de caso apresentado, os autores escolheram o esqueleto de paralelismo *Task Farm*, no qual não há interação entre as tarefas trabalhadoras. Para modelar esse exemplo, usaram um tratamento estatístico no tempo médio de execução de cada etapa do *Task Farm*. Também consideram que a adaptação

Figura 4.4: Unidade de execução paralela de acordo.



Fonte: Adaptado de (MENCAGLI; VANNESCHI; VESPA, 2013).

não poderia considerar o prazo completo da execução. Cada mudança tinha horizonte de apenas alguns passos no futuro. Isso serviu para limitar o custo da adaptação. A execução de simulações em um *cluster* demonstrou que a solução apresenta resultados próximos ao algoritmo ótimo e superior a heurísticas com estratégia de reação pura. Os autores planejam no futuro expandir o trabalho para controlar grafos de computação com múltiplos módulos paralelos com cada um exibindo uma estratégia de controle própria.

O ParMod e o CodeCloud encapsulam o código paralelo em componentes. Dizemos que essa abstração é de grossa granularidade. Enquanto no CodeCloud, o componente tarefa possui informações para guiar o gerenciador de recursos, no ParMod o próprio componente interage com a plataforma para realizar as ações de elasticidade. Nesse último caso, há um controle maior por parte do desenvolvedor. Entretanto, como não há o gerenciador de recursos, o mapeamento de cada componente para plataformas é feito de maneira estática pelo usuário e a elasticidade do componente fica restrita aos recursos da plataforma escolhida no início da execução.

4.2.3.3 *Elastic MPI*

As duas soluções apresentadas nesta seção tratam o código paralelo como componentes de grossa granularidade. As próximas soluções estão integradas aos modelos de programação paralela mais elementares, dizemos abstrações de fina granularidade. Considerando o modelo de troca de mensagens e a especificação MPI, o Elastic MPI considera uma aplicação como uma sequência de iterações (RAVEENDRAN; BICER; AGRAWAL, 2011). A cada número X de iterações, o ambiente proposto avalia o tempo médio de execução de uma iteração. Caso esse valor esteja acima do desejado pelo usuário, os processos

são terminados, os valores das variáveis (estado da computação) são salvos em arquivos e a computação é reiniciada com um número maior de processos (máquinas virtuais). O mesmo ocorre se o tempo de execução médio de uma iteração esteja muito abaixo do esperado pelo usuário. Porém, neste caso, a computação é reiniciada com menos processos (VMs) com o objetivo de poupar custos. A técnica é reativa, ou seja, não realiza nenhuma previsão sobre o comportamento da próxima iteração. Para essa abordagem funcionar, o desenvolvedor deve projetar a aplicação com um laço principal e definir as variáveis e estruturas de dados de maneira que elas possam ser salvas em disco e redistribuídas para os novos processos.

4.2.3.4 *Malleable MPI*

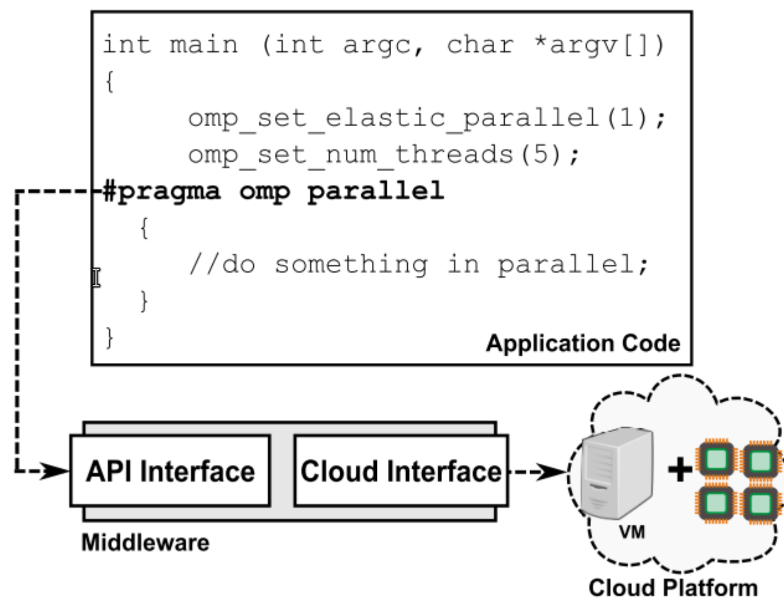
Também relacionado ao MPI, o ambiente descrito em Malleable MPI apresenta uma solução que permite alterar a granularidade de processos, além de migração dos mesmos (EL MAGHRAOUI, 2009). O objetivo é distribuir a carga de maneira homogênea e permitir que a aplicação aproveite novos recursos. As técnicas usadas são a migração de processos dentro do *cluster*, a partição (*split*) e a fusão (*merge*) de processos. O benefício da partição é distribuir a carga de um processo entre novos recursos. Já a fusão visa consolidar processos em uma mesma máquina, diminuindo o custo da troca de contexto. Cabe ao desenvolvedor adaptar o código MPI, utilizando chamadas que permitam informar ao sistema que faz o perfilamento dinâmico da aplicação. O sistema de tempo de execução é formado pelo serviço de *checkpointing* PCM e o ambiente par-a-par (*peer-to-peer*) IOS (*Internet Operating System*). A solução foi planejada considerando apenas ambientes de *cluster computing*. Portanto, as decisões de migração, partição e fusão são tomadas levando em consideração apenas a carga de cada processo. A qualidade da rede de interconexão não é considerada na migração, fator que precisa ser considerado quando a plataforma de execução é uma nuvem computacional.

4.2.3.5 *Cloudline*

Galante e Bona apresentam uma solução chamada Cloudline para elasticidade vertical (aumento de recursos dentro da máquina virtual) de aplicações escritas usando diretivas OpenMP (GALANTE; BONA, 2014). A solução apresentada é composta por dois módulos: uma API OpenMP estendida e um *middleware* de elasticidade (ver Figura 4.5).

A biblioteca OpenMP adapta as diretivas existentes e fornece algumas funções para lidar com a elasticidade. As diretivas adaptadas são a `parallel`, `single` e `sections`. O princípio observado em todas as diretivas é aumentar ou diminuir o número de CPUs virtuais (VCPU) a medida que o código entra em regiões paralelas ou as conclui. As funções

Figura 4.5: API e *Middleware*.



Fonte: Adaptado de (GALANTE; BONA, 2014).

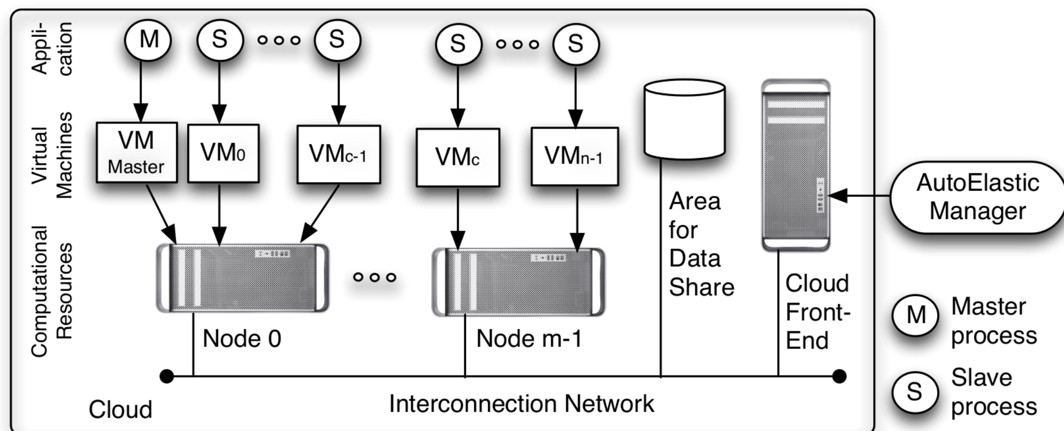
são opcionais, permitindo ao usuário um controle mais fino da elasticidade. A exceção fica por parte da função `omp_alloc`, que permite aumentar ou diminuir a quantidade de memória alocada à máquina virtual. O *middleware* de elasticidade é uma camada que traduz as diretivas e funções da API em comandos para a nuvem. Funções foram também acrescentadas na API para suporte à elasticidade horizontal (GALANTE; BONA, 2015), mas não demonstraram integração com o padrão MPI. No entanto, os experimentos mostram que a arquitetura é adequada para aplicações mestre escravo ou *bag of tasks*.

4.2.3.6 AutoElastic

AutoElastic constitui um ambiente que monitora a carga de máquinas virtuais que executam uma aplicação paralela no padrão mestre/escravo e toma decisões de elasticidade baseadas no comportamento da aplicação (R. RIGHI, 2016). O diferencial é que o monitoramento de carga é feito baseado em uma técnica de "envelhecimento"(*aging*). Trata-se de um aprimoramento de um laço de decisão que atribui mais peso aos acontecimentos recentes e elimina falsos positivos ou negativos.

Na Figura 4.6, temos a distribuição dos elementos na arquitetura da AutoElastic. O processo que gerencia a elasticidade (*AutoElastic Manager*) executa em um recurso computacional externo à nuvem. O gerente emite comandos para o *Front End* da nuvem para controlar os recursos e interage com o processo mestre da aplicação para adaptá-la. Essa comunicação é feita através da atualização de estruturas em sistema de armazenamento

Figura 4.6: Arquitetura da AutoElastic.



Fonte: Adaptado de (R. RIGHI, 2016).

compartilhado. A informação que o gerente recupera do ambiente representa a distribuição de carga entre as máquinas virtuais. Esses dados servem como entrada para um conjunto de regras e ações definidas pelo usuário que guiam o processo de elasticidade. A natureza do mecanismo é semelhante ao laço de decisão descrito na Seção 4.1.3.1.

Como pode ser visto no esquema da Figura 4.6, as aplicações suportadas pela AutoElastic devem ser estruturadas de acordo com o padrão mestre/escravo. Por exemplo, no trabalho citado, os autores avaliam a solução através de uma aplicação para calcular a integral de uma função em um dado intervalo. Eles definiram um processo mestre responsável por divisão do intervalos em seções para cada processo e um processo escravo para calcular a integral no intervalo. O ambiente fornece métodos para que o mestre possa controlar a execução global e que permitem aos escravos recuperar sua fatia do trabalho e notificar o mestre.

4.2.3.7 *Análise Comparativa*

Sobre os trabalhos discutidos, podemos afirmar que o Elastic MPI, Malleable MPI e o Cloudline são usados para desenvolver os componentes maleáveis utilizados no CloudCode, ParMod e AutoElastic. As três primeiras soluções atuam em um nível mais próximo da visão do desenvolvedor, enquanto a AutoElastic é uma abstração intermediária e o CloudCode e o ParMod adotam um modelo de encapsulamento do código. Dada a variedade de aplicações científicas, suportar mais de uma técnica de elasticidade seria uma funcionalidade interessante, entretanto nenhuma das soluções dá suporte a essa customização.

A Tabela 4.1 apresenta uma visão geral do suporte à elasticidade no nível de programação. Algumas técnicas das aplicações comerciais podem ser utilizadas na Computação

Tabela 4.1: Visão geral da elasticidade em CAD a nível do desenvolvedor.

Solução	**Modelo de Programação**	**Granul. do Código**	**Tipo de Elasticidade**	**Técnica de Decisão**	**Adapt. em Execução**	**Execução Multiplat.**
CodeCloud	Vários	Alta	H	Regras	Não	Sim
ParMod	Vários	Alta	H	T. do Controle	Sim	Sim
Elastic MPI	MPI	Baixa	H	An. da Carga	Sim	Não
Malleable MPI	MPI	Baixa	H	An. da Carga	Sim	Não
Cloudline	OpenMP+MPI	Baixa	H/V	Desenvolvedor	Sim	Não
AutoElastic	Mestre/Escravo	Média	H	Regras	Sim	Não

Fonte: Elaborada pelo autor.

de Alto Desempenho, mas aqui a análise temporal da carga trata do uso de CPU nos nós de processamento, não da carga de entrada ou tamanho do problema. A maioria das soluções trata da elasticidade horizontal, o que é esperado já que nem todos os *hypervisors* suportam elasticidade vertical. Como consequência, aplicações MPI ocupam posição de destaque. Importante observar a diversidade das técnicas de decisão adotadas, o que nos leva a crer que uma única técnica não é capaz de representar todo o espectro das diversas aplicações de Computação de Alto Desempenho.

4.3 Conclusões

Neste capítulo, apresentamos os conceitos de elasticidade e como são considerados no contexto da Computação de Alto Desempenho. As soluções existentes apresentam suporte a vários modelos de programação, mas consideramos que nenhuma permite controle total pelo desenvolvedor na definição de uma técnica de decisão customizada para cada tipo de computação. Através do monitoramento da execução da aplicação, sustentamos a hipótese de que permitir personalizar a tomada de decisão para cada aplicação será benéfica para todos os atores envolvidos. Mas para atingir tal objetivo, precisamos de um ambiente de desenvolvimento auxiliar que permita descrever computação e plataformas e raciocinar sobre os metadados disponíveis.

5. A Nuvem de Componentes HPC Shelf

A HPC Shelf é uma plataforma de computação em nuvem que fornece serviços para o desenvolvimento, implantação e execução de aplicações com requisitos de Computação de Alto Desempenho. O seu princípio básico é o desenvolvimento orientado a componentes, onde códigos de computação científica constituem *sistemas de computação paralela*, os quais são implementados através da composição de componentes que representam seus elementos de *software* e *hardware*, tanto funcionais quanto não-funcionais. Dentre esses elementos, estão incluídos plataformas de execução, algoritmos, estruturas de dados, esqueletos de programação, padrões de comunicação, estratégia de balanceamento de carga, e assim por diante. Para atingir tal flexibilidade, os componentes são descritos de acordo com o Hash (CARVALHO JUNIOR; LINS, 2008), um modelo de componentes paralelos que os classifica em diferentes espécies, que por sua vez representam conjuntos de componentes com modelos similares de implantação, execução e interação.

Outra característica marcante da HPC Shelf é a adoção de um sistema de *contratos contextuais* que, além de separar a implementação dos componentes das suas interfaces, permite controlar a escolha de componentes apropriados de acordo com seus contextos de implantação, representando tanto requisitos da aplicação quanto características da plataforma de computação paralela hospedeiras.

Para os propósitos desta Tese, os interesses de reconfiguração elástica são descritos utilizando a HPC Shelf. Dessa maneira, podemos reaproveitar os serviços da nuvem e nos concentrar em adicionar as funcionalidades de elasticidade que consideramos importantes. Neste capítulo, discutimos os principais conceitos da HPC Shelf, enquanto no Capítulo 6 apresentamos como os interesses de elasticidades são integrados à essa plataforma. A próxima seção apresenta uma visão abstrata para um conceito central para a HPC Shelf, os sistema de computação paralela, a qual será refinada nas seções posteriores, com a apresentação de outros conceitos relevantes.

5.1 Sistemas de Computação Paralela

Na plataforma HPC Shelf, um *sistema de computação paralela* é definido como uma composição de componentes paralelos extraídos de um catálogo de componentes mantido por essa plataforma. Destina-se a implementar uma solução computacional para um problema cujos requisitos de processamento exigem processamento paralelo de larga escala,

definido pelo emprego de múltiplas plataformas de computação paralela, possivelmente localizadas em domínios geográficos distintos e sob controle administrativo de instituições independentes.

Os elementos arquiteturais presentes em um sistema de computação paralela da HPC Shelf são dois componentes distintos, chamados Workflow (*componente de orquestração*) e Application (*componente de aplicação*), e um conjunto de *componentes de solução*. O componente de aplicação é responsável pela interação do sistema de computação paralela com uma aplicação da HPC Shelf. Por sua vez, o componente de orquestração coordena o ciclo de vida e guia a computação dos componentes de solução em direção à solução do problema computacional ao qual o sistema de computação paralela é destinado a resolver, através da ativação de *ações* computacionais exportadas por esses componentes.

O ciclo de vida de cada componente de solução é controlado pelo componente de orquestração através da ativação das seguintes ações computacionais:

- *ação de resolução* (**resolve**), para seleção de uma implementação do componente adequada para o contexto de execução atual, descrito através de seu *contrato contextual*, envolvendo os requisitos da aplicação em relação ao componente e as características da plataforma de computação paralela sobre a qual será instanciado.
- *ação de implantação* (**deploy**): para implantação do componente selecionado sobre a plataforma de computação paralela alvo, envolvendo possivelmente a sua compilação e ligação a bibliotecas.
- *ação de instanciação* (**instantiate**): para criação de uma instância do componente na plataforma de computação paralela alvo, de forma que se torne apta a execução de suas ações computacionais.

Alguns componentes de solução, ditos de computação, exportam uma ou mais ações de computação. Uma vez que são instanciados, essas ações podem ser ativadas pelo componentes de orquestração, guiando a computação do sistema de computação paralela.

Nos parágrafos que se seguem, o conceito de sistema de computação paralela introduzido nessa seção será refinado. Na Seção 5.2, é discutido o modelo de componentes paralelos Hash, o qual pressupõe a distinção dos componentes das plataformas que o suportam em um conjunto de espécies de componentes. Na Seção 5.3, são apresentadas as espécies de componentes da HPC Shelf, que delimitam os componentes de solução que podem ser empregados na composição de sistemas de computação paralela. Na Seção 5.4, são apresentados os papéis dos atores envolvidos na construção e utilização de sistemas de computação paralela, bem como de seus componentes. Na Seção 5.5, é discutido o conceito de contrato contextual, através do qual componentes são referenciados em sistema

de computação paralela a fim de que uma de suas implementações sejam selecionadas pelo sistema de resolução. Na Seção 5.6, são apresentados os elementos arquiteturais da HPC Shelf, oferecendo-se uma visão mais detalhada de como os sistemas de computação paralela são instanciados e executados sobre um conjunto de plataformas de computação paralela. Por fim, na Seção 5.7, é apresentado como a HPC Shelf lida com contratos contextuais enriquecidos com parâmetros de QoS (*quality of service*), que serão usados no mecanismo de reconfiguração elástica que motiva este trabalho.

5.2 O Modelo de Componentes Hash

A HPC Shelf emprega conceitos já aplicados com sucesso em ambientes de desenvolvimento baseados em componentes para Computação de Alto Desempenho. Nesta seção, contextualizamos o modelo Hash e sua implementação de referência, o HPE.

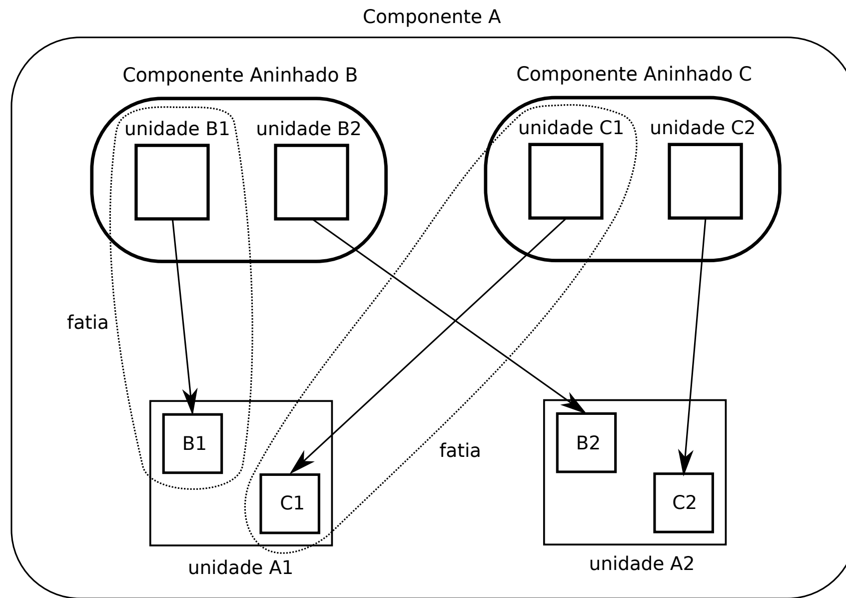
5.2.1 Componentes Paralelos e Composição por Sobreposição

O conjunto de componentes de uma plataforma de aplicações baseadas em componentes que adere ao modelo Hash possui três características:

- cada componente é formado por um conjunto de *unidades*, instanciadas em nós distintos de uma plataforma de computação paralela de memória distribuída;
- componentes podem ser combinados, hierarquicamente e recursivamente, através da *composição por sobreposição* (CARVALHO JUNIOR; LINS, 2008);
- os componentes encontram-se classificados conforme um conjunto de *espécies de componentes*, cada uma agrupando componentes que compartilham interesses específicos que exigem modelos distintos de implantação, execução e interação em relação a componentes de outras espécies.

Um interesse paralelo é definido como um interesse de *software* cuja implementação é transversal aos processos de um programa paralelo convencional. Computações paralelas são interesses paralelos funcionais, enquanto o mapeamento de processos em nós de processamento é um exemplo de um interesse paralelo não funcional. Um conjunto de unidades implementa um interesse paralelo encapsulado em um único componente Hash, representando, cada uma, o papel de um processo no tratamento do interesse. Por exemplo, ao distinguir processos através do *rank* em uma computação MPMD , é possível definir componentes que tratam apenas de um interesse, sem a necessidade distribuir a implementação de um interesse em vários componentes distintos.

Figura 5.1: Exemplo de composição por sobreposição.



Fonte: Elaborada pelo autor.

A composição por sobreposição pode ser vista como uma álgebra de composição de componentes (CARVALHO JUNIOR; LINS, 2008). Ela demonstra como construir um componente Hash através da combinação de outros componentes Hash, chamados de *componentes aninhados*. Podemos considerar a composição por sobreposição como uma função parcial que tem como domínio o conjunto das unidades dos componentes aninhados e, como imagem, as unidades do componente Hash que os engloba. Cada mapeamento é denominado uma *fatia* da unidade alvo. A Figura 5.1 apresenta uma exemplo da composição. O componente A é formado pelos componentes aninhados B e C. As unidades de A são formadas pelo mapeamento das unidades dos componentes aninhados. Apesar de não ser demonstrado na figura, os próprios componentes aninhados B e C também podem ter componentes aninhados, compostos por sobreposição.

5.2.2 HPE

O HPE (*Hash Programming Environment*) é a implementação usada como referência do modelo Hash, voltada para plataformas de computação paralela em *cluster* (CARVALHO JUNIOR, 2007) (CARVALHO JUNIOR; REZENDE, 2013). Sua importância no contexto deste trabalho deriva do fato de que muitos dos conceitos hoje adotados na HPC Shelf surgiram dessa plataforma. Para habilitar o suporte a computação paralela em arquiteturas baseadas em *clusters*, o ambiente HPE suporta as seguintes espécies de componentes:

- *aplicações*, representando programas paralelos capazes de executar em *cluster*, compostos por componentes das outras espécies;
- *computações*, representando a implementação de algoritmos paralelos capazes de explorar as características particulares de um *cluster* onde a aplicação irá executar e também de resolver o problema computacional exigido;
- *estruturas de dados*, representando estruturas de dados distribuídas sobre os nós de processamento de um *cluster*, de interesse das computações;
- *sincronizadores*, representando padrões ou meios de interação (comunicação e sincronização) entre as unidades de cada componente paralelo;
- *qualificadores*, representando características não-funcionais abstratas dos componentes paralelos;
- *plataformas*, representando os *clusters* (plataformas de computação paralela) onde as aplicações podem executar;
- *ambientes*, representando bibliotecas e ambientes de controle do paralelismo dentro de cada componente.

O HPE é constituído por três elementos arquiteturais:

- o *Front-End*, um ambiente integrado de desenvolvimento para construção de componentes das diferentes espécies, possivelmente via composição por sobreposição;
- o *Back-End*, oferecendo um serviço de implantação e execução de aplicações, bem como seus componentes, sobre um cluster onde encontra-se instalado;
- o *Core*, oferecendo um serviço de catalogação e seleção de componentes, através de um sistema de tipo de componentes chamado HTS (CARVALHO JUNIOR, 2016).

O HTS (*Hash Type System*) é o precursor do sistema de contratos contextuais da HPC Shelf. Permite a coexistência de diversas implementações de um mesmo componente, dito *componente abstrato*, cada uma otimizada para explorar diferentes possibilidades de características arquiteturais de *clusters* distintos, bem como atendendo a diferentes requisitos de aplicações hospedeiras distintas (contexto). Para isso, componentes são referenciados em outros componentes e aplicações através de *tipos de instanciação*, cuja resolução conduz à escolha de *componente concretos* para a aplicação. Componente concreto é o nome que se atribui às implementações de componentes abstratos para diferentes contextos.

O HPE pode ser também enxergado como um *framework* de componentes de propósito geral que segue o padrão CCA (CARVALHO JUNIOR; CORREA, 2010). Sob essa

perspectiva, o HPE oferece uma noção mais geral de componente paralelo para o padrão CCA, baseado no modelo Hash, que poderia ser reaproveitada por outros *frameworks* CCA (CARVALHO JUNIOR; REZENDE, 2013).

O HPE introduz uma útil distinção entre unidades *singulares* ou *paralelas*, herdada pela HPC Shelf. No caso de unidades singulares, apenas uma única instância da unidade é criada em um único nó de processamento na instanciação do componente. No caso de unidades paralelas, várias instâncias da unidade podem ser criadas em nós de processamento distintos, visto como um *regimento de unidades* que implementam uma computação ao estilo SPMD (*Single Program Multiple Data*).

5.3 As Espécies de Componentes da HPC Shelf

Espécies de componentes agrupam componentes Hash que compartilham os mesmos modelos de implantação, execução e interação, geralmente tratando de um certo conjunto de interesses em comum. As espécies de componentes da HPC Shelf são:

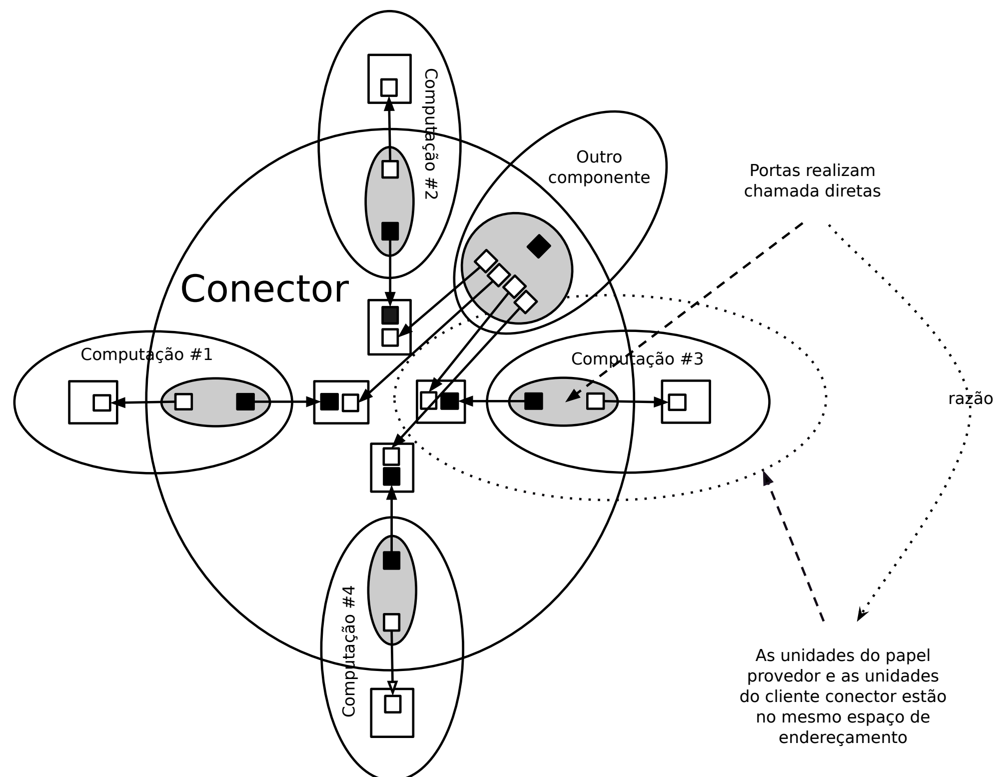
- *plataformas virtuais*, representando plataformas de computação paralelas de memória distribuída (e.g. *clusters* e MPPs [1]);
- *computações*, representando implementações de algoritmos paralelos capazes de tirar proveito de características arquiteturais peculiares de uma determinada classe de plataformas virtuais;
- *fontes de dados*, encapsulando grandes massas de dados que interessam às computações, acessíveis por meio de serviços locais ou remotos;
- *conectores*, os quais ligam componentes de computação e fontes de dados localizados em plataformas virtuais distintas, com diferentes propósitos;
- *ligações* (*bindings*), os quais ligam componentes entre si por meio de interfaces de serviços (*bindings de serviços*) ou de ações computacionais (*bindings de ações*);

Bindings acoplam componentes das demais espécies através da conexão de portas. Portas de *bindings* de serviços oferecem uma interface de serviços não-funcionais que podem ser usadas para acesso ao estado, dados, monitoramento e configuração de componentes. Uma porta de serviços suporta os papéis de usuária (*porta usuária*) ou provedora (*porta provedora*). O componente que fornece uma porta provedora disponibiliza um serviço que é utilizado por um componente que apresente uma porta usuária, desde que exista um *binding* compatível para os tipos dessas portas, os quais podem ser distintos,

[1] *Massive Parallel Processors*

quando o *binding* é responsável pela adaptação (ver Seção 3.1.1). Por sua vez, portas de *bindings* de ações servem para orquestração de ações funcionais entre os componentes de computação e conectores. Para isso, as portas de ações associadas por um *binding* devem possuir o mesmo conjunto de *nomes de ações*. Através de *bindings* de ações, componentes se tornam parceiros que sincronizam através de *rendezvous*, no qual os componentes envolvidos reagem à ativação síncrona de ações.

Figura 5.2: Conector na nuvem HPC Self.



Fonte: Elaborada pelo autor.

Na Figura 5.2, é apresentado um exemplo de um componente conector que acopla quatro componentes de computação. O objetivo desse exemplo é mostrar o conceito de *faceta* de conector. Nesse conector, o acoplamento às computações é feita por meio de *bindings* de serviços cujas portas usuárias estão do lado dos componentes computação e portas provedoras do lado do componente conector. Dessa forma, esse conector atua como um meio através do qual os componentes de computação podem sincronizar e/ou comunicar-se, ou seja, executam uma coreografia de uma interação. Note que cada componente de computação possui uma única unidade paralela e o conector possui uma unidade paralela que será ligada, através do *binding*, a cada unidade de um dos componentes de computação.

Através da porta usuária, as unidades dos componentes de computação podem realizar invocações ao serviço do *binding*, o qual as encaminharão ao conector. Nesse caso, os *bindings* são diretos, ou seja, suas unidades servidoras e provedoras executam no mesmo espaço de memória, onde estão localizadas a unidade do componente de computação e a unidade correspondente do conector. Para que isso seja possível, as unidades de um conector encontram-se distribuídas em diferentes plataformas virtuais, aonde estão localizados os componentes a ele ligados. Cada subconjunto de unidades de um conector que encontra-se em uma plataforma virtual distinta é chamada de faceta. É então responsabilidade do conector gerenciar a sincronização e comunicação entre as suas facetas, através de uma interface de troca de mensagens.

De acordo com a natureza das portas, um conector pode ter diferentes propósitos ao acoplar componentes de computação e fontes de dados. Os propósitos fundamentais são a *orquestração* e a *coreografia*. Na orquestração, através de portas de ações, o conector coordena a execução de um conjunto de ações computacionais de componentes de computação. Na coreografia, como no exemplo hipotético apresentado, a interação entre os componente é realizada através de *bindings* de serviço. É possível também especificar configurações híbridas de orquestração e coreografia, combinando *bindings* de serviço e de ações. Dessa forma, a espécie conector fornece a flexibilidade necessária para representar os diversos padrões de comunicação e interação existentes entre componentes paralelos.

Cada espécie representa um interesse diferente. Devido ao ambiente encontrado nas aplicações de computação científica, papéis distintos, podem ser atribuídos para cada usuário da plataforma de acordo com o interesse relevante. Para representar essa diversidade, definimos os atores da HPC Shelf na próxima seção.

5.4 Atores Envolvidos

A HPC Shelf prevê um ambiente no qual os atores envolvidos, com interesses complementares atuam para oferecer serviços de Computação de Alto Desempenho a *usuários especialistas*. Esses usuários, que constituem a primeira classe de atores, estão interessados em utilizar uma interface de alto nível para especificar os problemas que desejam solucionar. Eles não manipulam componentes diretamente, que estão ocultos atrás das abstrações de uma interface de programação declarativa específica de domínio.

Os *provedores de aplicações* desempenham fornecem aplicações para os usuários especialistas, implementando abstrações para problemas e os mapeando em sistemas de paralelos que irão resolvê-los. Para atingir esse objetivo, eles são especialistas em soluções computacionais no domínio, sendo capazes de compor os componentes apropriados que formam os sistemas de computação paralela que interessam a uma aplicação.

Os *desenvolvedores de componentes* programam os componentes que aplicações utilizam para compor sistemas de computação paralela. Componentes tendem a ser independentes da aplicação, já que desenvolvedores estão interessados em fornecer componentes para vários provedores. Além do mais, já que componentes são configurados para otimizar o desempenho em uma dada classe de plataformas virtuais, presume-se que desenvolvedores sejam especialistas em programação paralela.

Em um nível mais baixo, os *mantenedores de infraestrutura* estão preocupados em fornecer os componentes que representam plataformas virtuais, que são instanciados sobre a infraestrutura que gerencia. Já que plataformas virtuais são tratadas como componentes (de *hardware*), os mantenedores devem registrá-los no mesmo catálogo no qual os desenvolvedores registram seus componentes.

Os atores envolvidos no desenvolvimento de componentes estão preocupados com interesses diferentes, mas precisam de uma representação em comum para descrição de componentes, de forma independente de como são implementados e permitindo a coexistência de múltiplas implementações de um mesmo componentes para diferentes combinações entre requisitos da aplicação e características da plataforma virtual alvo. Os contratos contextuais servem como esquema de anotação de metadados e permitem que os serviços da HPC Shelf comuniquem-se sobre artefatos de *software* distintos.

5.5 O Sistema de Contratos Contextuais

O objetivo da Computação de Alto Desempenho é atingir o máximo do processamento potencial das infraestruturas de computação paralela disponíveis. Em uma abordagem orientada a componentes, para que isso seja possível, os componentes devem ser desenvolvidos considerando informações detalhadas sobre a arquitetura das plataformas que formam tais infraestruturas. Uma mesma funcionalidade pode ser oferecida por um componente especializado para plataformas de computação paralela de memória compartilhada e por outro componente desenvolvido para plataformas de memória distribuída.

A escolha do componente depende do contexto do ambiente sobre o qual o componente executará. Para os provedores de aplicações, que trabalham em um nível de abstração mais próximo do domínio de conhecimento dos usuários especialistas, realizar a escolha adequada das plataformas e componentes para tratar determinado problema não é uma tarefa trivial. É necessário definir um formalismo para que mantenedores e desenvolvedores possam descrever as características de suas plataformas e componentes de computações, permitindo o estabelecimento de uma base de dados para a consulta dos provedores de aplicação.

Na HPC Shelf, o sistema de contratos contextuais apresenta as funcionalidades necessárias para auxiliar na escolha de componentes das diferentes espécies de componentes que suporta. Para descrever um componente, o primeiro passo é definir um esquema para representar as informações importantes sobre o artefato. Para tal, um *componente abstrato* descreve um conjunto de propriedades que direcionam a forma como são implementados um conjunto de componentes na HPC Shelf relacionados entre si por um interesse comum que desejam realizar para usufruto das aplicações. O conjunto de propriedades é representado por meio de uma abstração de contexto, especificado por um conjunto de *parâmetros de contexto*. Para exemplificar o conceito de componente abstrato, considere a representação abaixo:

MatrixMultiplication
[***matrix_type*** = A: MatrixType, ***sparsity*** = B: RealNumber,
platform = [***platform_type*** = C: PlatformType, ***processor_type*** = D: CPUType]]

O componente abstrato MatrixMultiplication representa o conjunto de componentes paralelos para multiplicações de matrizes. Nesse esquema, estão parâmetros que representam características da plataforma que executa a computação e do formato da entrada que é submetida ao componente. Por exemplo, ***matrix_type*** é um parâmetro de contexto que determina que tipo de matriz é submetido para multiplicação. Esse parâmetro permite escolher componentes concretos otimizados para cada tipo de matriz. No componente abstrato, uma variável de contexto é atribuída ao parâmetro, sendo que essa variável é limitada por um *qualificador*. Continuando no exemplo, a variável A, atribuída ao parâmetro ***matrix_type***, é limitada pelo qualificador MatrixType.

Quando um provedor de aplicação oferece argumentos aos parâmetros de um componente abstrato, temos um *contrato contextual*. Dessa forma, um contrato contextual é definido pela associação de um componente abstrato a um conjunto de argumentos de contexto, cada qual atribuído a um dos parâmetros de contexto do componente abstrato. Abaixo, o contrato contextual MatrixMultiplicationMPI preenche com argumentos os parâmetros do componente abstrato MatrixMultiplication:

MatrixMultiplicationMPI
[***matrix_type*** = SquareMatrix, ***sparsity*** = 0.1,
platform = [***platform_type*** = Cluster, ***processor_type*** = X_86_64_CPU]]

O parâmetro ***matrix_type*** recebe como argumentos o qualificador SquareMatrix. Portanto, o provedor deseja um componente otimizado para matrizes quadradas. O algoritmo de resolução do HPC Shelf, ao receber um contrato, propaga os argumentos para os componentes aninhados e através da criação de uma árvore estruturando os tipos dos

componentes catalogados, retorna os componentes concretos que suportam o contexto requisitado, através de uma relação de tipos e subtipos, onde contratos contextuais denotam tipos de componentes.

No exemplo mencionado sobre multiplicação de matrizes, tratamos de um componente abstrato da espécie computação. Mas devemos lembrar que na HPC Shelf, plataformas virtuais também são componentes. Na representação no sistema de tipos, uma *categoria de plataforma virtual* é um componente abstrato utilizado para representar uma classe de plataformas de computação paralela. Por exemplo, podemos ter a categoria de plataformas que representam *clusters*. Para esse componente abstrato, um possível parâmetro de contexto é o número de nós de processamento. O contrato contextual que preenche os parâmetros de componente abstrato de uma plataforma é chamado de *perfil de plataforma virtual*, contendo as informações necessárias para criar uma plataforma virtual que atende às restrições do contexto. Por exemplo, considere os seguintes componentes abstratos para representar os elementos que compõem um *cluster*:

PROCESSOR [***number_of_cores*** = I: INTEGERNUMBER, ***clock_per_core*** = R: REALNUMBER, ***cache_memory*** = J: INTEGERNUMBER]

MEMORY [***quantity*** = I: INTEGERNUMBER, ***frequency*** = R: REALNUMBER, ***technology*** = T: MEMORYTYPE]

NETWORK [***latency*** = I: INTEGERNUMBER, ***bandwidth*** = J: INTEGERNUMBER, ***topology*** = T: TOPOLOGYTYPE, ***technology*** = R: NETWORKTYPE]

CLUSTER [***nodes*** = I: INTEGERNUMBER, ***processor*** = P: PROCESSOR, ***network*** = N: NETWORK]

Mostramos apenas os componentes principais e simplificamos os parâmetros de contexto para efeito didático. Temos no componente abstrato CLUSTER uma categoria de plataforma na qual encontramos a cardinalidade dos nós de processamento e um parâmetro definindo o tipo desse mesmo nó. A partir do componente de mais alto nível, percorrendo as definições dos tipos encontramos as características de *hardware* que são responsáveis por determinar o desempenho do computador paralelo. Entretanto, o esquema apresentado não é definitivo e pode ser estendido da maneira que o mantenedor desejar. Por exemplo, nada impede que um mantenedor defina um *cluster* com dois tipos de nó de processamento, contendo uma quantidade diferente de cada um deles. Mesmo dentro da definição de um nó, podemos ter vários tipos de armazenamento e redes de interconexão. Apesar de no exemplo descrevermos um *cluster* físico, podemos utilizar a mesma metodologia para descrever os perfis de *clusters* virtuais disponíveis em nuvens públicas.

Preenchendo cada parâmetro com argumentos de contextos que definem as características de uma plataforma virtual, o mantenedor disponibiliza para os outros atores que

tipo de plataforma virtual pode ser instanciada na infraestrutura que mantém. Com a descrição das computações e plataformas, os serviços da HPC Shelf podem atuar no controle do ciclo de vida da execução de aplicações de Computação de Alto Desempenho. Como exemplo de um perfil de plataforma baseado nos componentes abstratos citados, temos os seguintes contratos contextuais:

XeonE5640:PROCESSOR [***number_of_cores*** = 8, ***clock_per_core*** = 2.67, ***cache_memory*** = 12]

Four8GbSticksMemoryCluster:MEMORY [***quantity*** = 24, ***frequency*** = 1066, ***technology*** = DDR3]

GigabitEthernet:NETWORK [***latency*** = 143, ***bandwidth*** = 1000, ***topology*** = StarTopology, ***technology*** = EthernetType]

AcademicCluster:CLUSTER [***nodes*** = 48, ***processor*** = XeonE5640, ***network*** = GigabitEthernet]

A resolução de contratos contextuais de componentes de computação em sistemas de computação paralela é realizada de forma concomitante à resolução das plataformas virtuais sobre as quais devem executar. Chamamos de componente *sistema computacional*, o par formado por um componente de computação e o componente plataforma ao qual ele encontra-se associado, ambos representados por contratos contextuais complementares, uma vez que os argumentos de contexto associados ao componente plataforma especializam os parâmetros de contexto associados ao componente de computação que delimitam a classe de plataformas virtuais para a qual foi desenvolvido para melhor explorar suas características.

O mecanismo de resolução de um contrato contextual da HPC Shelf executa em duas fases: *seleção* e *classificação*. Na primeira, são selecionados os sistemas computacionais (componente de computação e plataforma virtual) que atendem ao contrato contextual, ou seja que respeitam os requisitos impostos pela aplicação (critério de segurança). Na última, os componentes selecionados são classificados segundo critérios de qualidade, determinados por parâmetros de contexto que descrevem requisitos de QoS[2] da aplicação, discutidos posteriormente, na Seção 5.7.

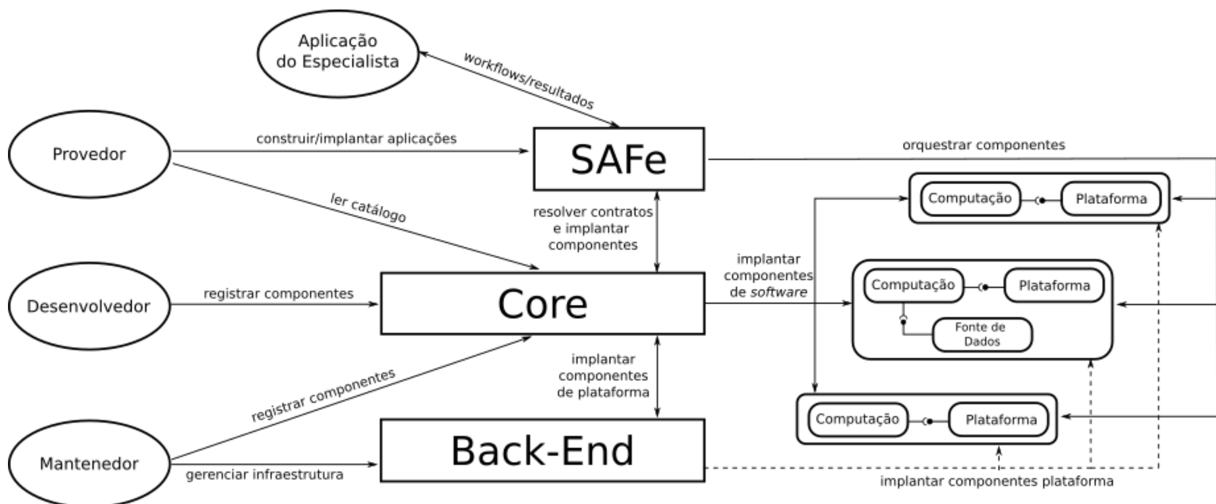
5.6 Arquitetura da HPC Shelf

Para atender aos interesses dos atores envolvidos, a HPC Shelf é formada por três elementos arquiteturais que comunicam-se entre si através da oferta de serviços, de forma

[2] *Quality of Service*

análoga ao HPE, chamados de Front-End, Core e Back-End, onde o Front-End é um *framework* de aplicações chamado SAFe, que também pode ser visto como um *sistema gerenciador de workflows científicos* (SWfMS) (SILVA; CARVALHO JUNIOR, 2016). A Figura 5.3 apresenta a arquitetura geral, com os papéis de cada ator e descrevendo suas interações com cada um dos serviços.

Figura 5.3: Arquitetura da HPC Shelf.



Fonte: Elaborada pelo autor.

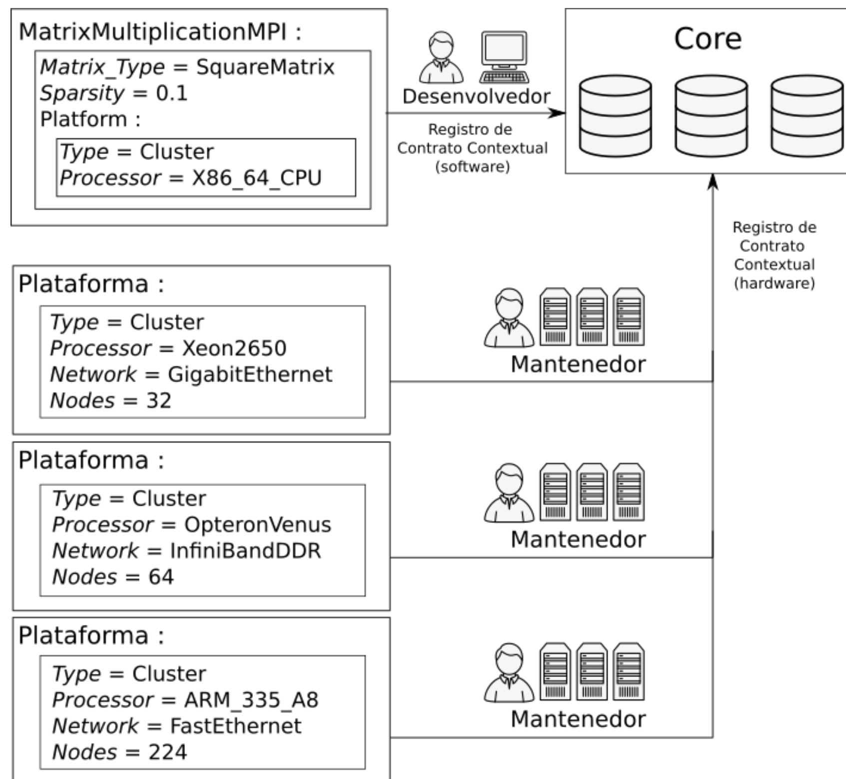
A partir do SAFe, provedores derivam aplicações orientadas ao usuário especialista, ocultando a abstração dos componentes. Para tal, o *framework* fornece acesso ao catálogo de componentes e os utiliza para criar *sistemas de computação paralela* formados por um conjunto de componentes das espécies suportadas pela HPC Shelf. Através de *workflows*, sistemas de computação paralela orquestram ações de componentes a fim de oferecer soluções para problemas de interesse da aplicação, especificados pelo usuário especialista.

Sistemas de computação paralela são especificados em uma linguagem chamada SAFeSWL (*SAFe Scientific Workflow Language*), baseada em XML. Essa linguagem possui um subconjunto arquitetural e outro subconjunto de orquestração, essa última constituindo-se em uma linguagem de *workflows*. Através do primeiro, os contratos contextuais dos componentes do sistema de computação paralela são especificados, bem como as ligações entre as instância de componentes envolvidas. Através do último, as ações de computação dos componentes são orquestradas.

O Core é um artefato que implementa um conjunto de serviços para gerenciar o ciclo de vida dos componentes, desde o seu registro em um catálogo de componentes até a sua implantação em plataformas virtuais na execução de sistemas de computação paralela. Portanto, ele gerencia o catálogo de componentes da HPC Shelf, implementando o sistema de contratos contextuais apresentados na Seção 5.5. Dessa forma, aplicações acessam o Core para resolver contratos contextuais especificados nos sistemas de computação paralela

e implantar os componentes selecionados nas plataformas virtuais selecionadas. Por sua vez, desenvolvedores e mantenedores registram contratos e componentes no catálogo, como demostrado na Figura 5.4. O processo de resolução implementado pelo Core envolve encontrar todas as possibilidades de sistemas computacionais concretos para um dado contrato contextual e ordená-las de acordo com parâmetros de qualidade previamente definidos. O SAFe recebe a lista ordenada de opções e tem liberdade para escolher qual sistema deve ser implantado pelo Core. A interação entre o Core e o SAFe é exemplificada na Figura 5.5.

Figura 5.4: Registro de Componentes no Core.



Fonte: Elaborada pelo autor.

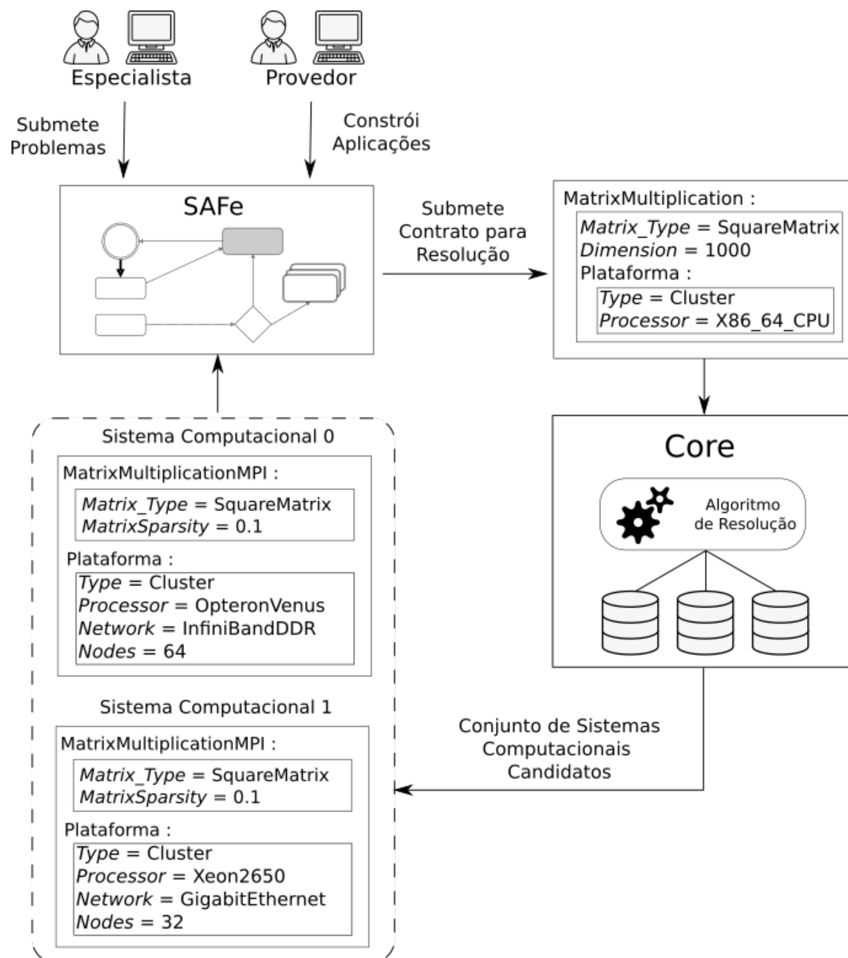
O Back-End é o serviço disponibilizado para oferecer as infraestruturas de computação paralela de cada mantenedor para o SAFe, com intermediação do Core, que o invoca para instanciar plataformas virtuais sobre essas infraestruturas. Uma vez que as plataformas virtuais são instanciadas, o Core pode implantar no sistema computacional as outras espécies de componentes, tais como computações, fontes de dados, conectores e ligações.

Após a implantação, a comunicação entre o *workflow* e os componentes é direta, sem a intermediação do Core. É no nível do sistema computacional paralelo criado pelo Back-End sobre a infraestrutura que devem ser instanciadas as unidades dos componentes de acordo com o modelo Hash, respeitando as regras da composição por sobreposição. Também cabe ao Back-End implantar de maneira correta as facetas dos conectores que permitem

a conexão entre componentes, estejam eles em um mesmo sistema computacional ou conectados a plataformas virtuais distintas.

Importante salientar que a HPC Shelf permite o reuso de *software* em diferentes níveis de abstração. Para o provedor, um modelo de sistema de computação pode ser criado utilizando componentes abstratos disponíveis no Core, onde um mesmo modelo pode ser encaminhado para resolução com diferentes argumentos de contexto para os parâmetros de cada componente abstrato. Na separação dos subconjuntos arquitetural e de orquestração da SAFeSWL, uma mesma arquitetura de sistema de computação pode ser executada com vários *workflows*. Já o desenvolvedor de componentes pode compor novos componentes a partir daqueles já descritos no catálogo, através da composição por sobreposição.

Figura 5.5: Resolução de Componentes pelo Core.



Fonte: Elaborada pelo autor.

5.7 Qualidade de Serviço em Contratos Contextuais

A resolução de contratos contextuais implementa critérios de corretude funcionais e não-funcionais para seleção de componentes. Tratam-se de *critérios de segurança*, que garantem que o componente selecionado atenderá aos requisitos da aplicação e são capazes de explorar as características das plataformas virtuais alvo do sistema de computação paralela. Mas qual dos sistemas computacionais presentes na lista de candidatos é o mais adequado para atender as expectativas da aplicação? Para responder a essa pergunta, introduzimos critérios de qualidade no sistema de resolução, adicionando parâmetros de QoS (*Quality of Service*) aos contratos.

Como discutido no Capítulo 3, um ambiente para Computação de Alto Desempenho baseado em componentes deve lidar com o *desempenho* e *acurácia* de componentes ligados à plataformas para garantir a Qualidade de Serviço Computacional (CQoS - *Computational Quality of Service*). De fato, nós avaliamos desempenho como o tempo de execução de componente de computação para uma determinada entrada. Por sua vez, a acurácia permite definir o quão perto a solução obtida por um componente está de um resultado desejado. Em geral, acurácia é um valor numérico fixo que corresponde ao percentual de fidelidade garantido, apesar desse valor poder variar durante a execução para componentes que implementam algoritmos aproximativos. Também consideramos a *eficiência*, representando o nível de utilização dos recursos, como outro parâmetro a ser analisado. Nós adotamos a carga média dos nós de processamento em um *cluster* como um exemplo do nível de eficiência. Por último, incluímos o *custo monetário* e o *consumo energético* como parâmetros, visto que são relevantes no contexto das nuvens computacionais e consciência ecológica. O custo se refere à quantidade de dinheiro que especialistas devem repassar aos mantenedores pelos recursos alocados durante a execução, enquanto o consumo mede a energia elétrica consumida em kWh.

Os desenvolvedores são responsáveis por informar parâmetros de QoS dos componentes computacionais nos contratos contextuais, já que compreendem o código e os algoritmos empregados. Para tal, o desenvolvedor informa uma *função de estimativa* anexada ao contrato. Essa função recebe como entrada os parâmetros da computação e da plataforma que compõem um sistema e retorna os valores numéricos para cada parâmetro de QoS. A etapa de classificação do procedimento de resolução no Core aplica a função de estimativa em cada sistema candidato e retorna uma lista ordenada para a aplicação.

Criar uma função de estimativa não é uma tarefa trivial. Considerando requisitos de desempenho, modelos analíticos preditivos (MARIN; MELLOR-CRUMMEY, 2004) são uma maneira de estudar o comportamento de uma computação através de um conjunto de equações que quantificam o total de trabalho realizado para cada nó de processamento,

considerando o padrão de acesso à memória, sobrecarga da comunicação e outras métricas. Essa abordagem usa metodologia semelhante às funções de isoeficiência discutidas na Seção 2.4. Outras abordagens empregam técnicas de aprendizagem de máquina (IPEK, 2005). A HPC Shelf apenas supõe que tal função de estimativa retorne um vetor de argumentos com uma entrada para cada parâmetro de QoS. Por exemplo, para um dado sistema computacional (CSC), formado por um contrato de plataforma (PC) e outro contrato para computação (CC), temos que:

$$QoS = estimation_function(CSC[PC, CC]) \tag{5.1}$$

O vetor $QoS = [a, e, t, c, p]$ tem um valor para acurácia, eficiência, tempo de execução, custo e consumo energético, respectivamente. O desenvolvedor está livre para escolher entre modelos analíticos, simulações, aprendizado de máquina, etc. O objetivo da estimativa é prover um guia geral que permite a HPC Shelf adaptar a execução de sistemas de computação paralela em busca de cumprir os requisitos de QoS. De fato, se a aplicação faz uma boa escolha para um sistema computacional inicial, é mais fácil adaptá-lo para atingir o estado de execução mais próximo dos requisitos. Esta é a razão pela qual a HPC Shelf deve fornecer uma maneira de ordenar os sistemas candidatos.

Nós definimos uma abordagem particular para classificar componentes candidatos que será adotada nesta Tese. Seja $CSC_i[PC_i, CC_i]^{\ i=1...n}$ o conjunto de componentes de sistema retornados pela etapa de seleção da resolução. Os valores retornados pelas funções de estimativa fornecidas pelo desenvolvedor de cada componente computação interno ao sistemas são armazenados no vetor $QoS_i = [a_i, e_i, t_i, c_i, p_i]$. Esse vetor é dividido em dois:

$$QoS_i^{max} = [a_i, e_i] \qquad\qquad QoS_i^{min} = [t_i, c_i, p_i] \tag{5.2}$$

QoS_i^{max} tem o valor para cada parâmetro de contexto de QoS com *subtipagem aditiva* (quanto maior o valor, melhor), enquanto QoS_i^{min} tem valores para os parâmetros com *subtipagem subtrativa* (quanto menor o valor, melhor). Seja $max(x)$ e $min(y)$, para $x \in \{a, e\}$ e $y \in \{t, c, p\}$, o valor máximo e mínimo para cada parâmetro de contexto de QoS, seja ele aditivo ou subtrativo. Podemos normalizar o vetor da seguinte forma:

$$QoS_i^{max} = [\frac{a_i - min(a)}{max(a) - min(a)}, \frac{e_i - min(e)}{max(e) - min(e)}] \tag{5.3}$$

$$QoS_i^{min} = [\frac{t_i - min(t)}{max(t) - min(t)}, \frac{c_i - min(c)}{max(c) - min(c)}, \frac{p_i - min(p)}{max(p) - min(p)}] \tag{5.4}$$

Provedores de aplicação podem informar dois novos vetores com pesos para cada parâmetro de QoS. O objetivo é definir a prioridade entre os parâmetros. Por exemplo, um

provedor pode priorizar tempo de execução ao invés de custo. Desta forma, o mecanismo de classificação retorna a lista de candidatos com os sistemas de maior desempenho próximo do topo. Novamente, o vetor tem o formato $W = [w_a, w_e, w_t, w_c, w_p]$, dividido de acordo com a tipagem:

$$W_{max} = [w_a, w_e], w_a + w_e = 1 \quad (5.5)$$

$$W_{min} = [w_t, w_c, w_p], w_t + w_c + w_p = 1 \quad (5.6)$$

Definimos o valor de ranqueamento para cada sistema $CSC_i[PC_i, CC_i]$ como:

$$R_i = (w_a * a_i + w_e * e_i) + (1 - (w_t * t_i + w_c * c_i + w_p * p_i)) \quad (5.7)$$

Os componentes de sistema computacionais paralelos candidatos são ordenados de acordo com o valor decrescente de $R_i\ ^{i=1...n}$, sendo que a aplicação escolhe o primeiro candidato na lista. Como um exemplo do método de classificação proposto, considere três componentes de sistema candidatos com os seguintes valores para os parâmetros de QoS:

$$QoS_0^{min} = [0.7, 0.8], QoS_0^{max} = [1600, 10, 3]$$
$$QoS_1^{min} = [0.9, 0.9], QoS_1^{max} = [1800, 6, 4]$$
$$QoS_2^{min} = [0.9, 0.7], QoS_2^{max} = [1500, 15, 5]$$

Após o processo de normalização, temos:

$$QoS_{0max} = [0, 0.5], QoS_{0min} = [0.34, 0.45, 0]$$
$$QoS_{1max} = [1, 1], QoS_{1min} = [1, 0, 0.5]$$
$$QoS_{2max} = [1, 0], QoS_{2min} = [0, 1, 1]$$

Se a aplicação fornecer o vetor de pesos $W = [0.7, 0.3, 0.8, 0.1, 0.1]$, priorizando acurácia e desempenho (tempo de execução), temos $R_0 = 0.84$, $R_1 = 1.15$, $R_2 = 1.50$. Podemos observar que o candidato com maior valor para R, (CSC_2), é aquele com maior acurácia (0.9) e menor tempo de execução (1500 segundos). Portanto, estará no topo da lista retornada à aplicação.

5.8 Conclusões

Através da HPC Shelf, os diversos interessados na execução de aplicações de alto desempenho têm um ambiente capaz de gerenciar infraestruturas e computações fornecendo a possibilidade de reutilizar esforços anteriores e com isso aumentar a produtividade de todos os envolvidos. Apesar do conceito de plataforma virtual permitir abstrair detalhes da infraestrutura na qual os componentes executam, seja ela uma estrutura física como um *cluster* ou um ambiente virtualizado em uma nuvem, até o momento o projeto da HPC Shelf só permite configurar a plataforma virtual no momento da instanciação do sistema computacional. Durante a execução, não há nenhuma adaptação da plataforma virtual e dos componentes a ela conectados pela parte de alocação. Dessa forma, as características de elasticidade nas nuvens e os serviços de alocação dos *clusters* tradicionais que permitem alterar os recursos em tempo de execução não são aproveitadas.

No Capítulo 6, partindo do conhecimento já estabelecido sobre a HPC Shelf, apresentamos soluções para adaptar o sistema computacional de acordo com mudanças no contexto das infraestruturas que compõem a nuvem de componentes.

6. Um Arcabouço para Construção de Aplicações de Computação de Alto Desempenho Elásticas

Como discutido no Capítulo 2, partimos do pressuposto de que o pesquisador que faz uso de Computação de Alto Desempenho tem à sua disposição diversas infraestruturas de computação paralela. Essas plataformas são compartilhadas com outros pesquisadores, sendo que o estado de seus recursos varia durante a execução da aplicação. Para obter melhor desempenho e maior satisfação do usuário, a execução do código deve ser adaptada às mudanças no estado dos recursos das infraestruturas.

No Capítulo 4, apresentamos trabalhos que empregam a reconfiguração elástica como uma solução para adaptação dentro de uma plataforma de computação paralela. Apesar de algumas soluções já apresentarem o conceito de componentes (ParMod, ver Seção 4.2.3.2), a plataforma não é representada como uma espécie de componentes e a maioria das soluções apenas definem interfaces de controle estáticas consolidadas em bibliotecas e serviços de gerência de infraestrutura com pouca flexibilidade. Dessa forma, não é tarefa trivial reutilizar os artefatos criados para definir novas políticas de reconfiguração e mecanismos de adaptação personalizados para cada aplicação.

Defendemos que para oferecer uma solução adequada às necessidades dos usuários, a representação tanto dos artefatos de *software* quanto dos recursos de *hardware* através da abstração de componentes cria mais possibilidades para o reuso. A HPC Shelf já apresenta essa capacidade, sendo que através dela podemos modelar as reconfigurações elásticas através da interação entre componentes, aprimorando a flexibilidade da adaptação já encontrada nas soluções baseadas em componentes discutidas no Capítulo 3. Nossa contribuição é fornecer um *framework* que representa todos os artefatos componentes ao mesmo tempo que trata a *reconfiguração elástica*, utilizando contratos contextuais para representação de composições de sistemas computacionais.

Para implementar o *framework* proposto, é necessário um conjunto de serviços que permitam armazenar informações estruturadas sobre plataformas e aplicações, além de fornecer operações para a instanciação de código. Este trabalho se enquadra no contexto da HPC Shelf, ambiente que adota o paradigma de programação baseada em componentes para representar tanto o *software* quanto o *hardware* engajado em computação paralela, representando com fidelidade o cenário atual do usuário com acesso a várias infraestrutu-

ras. Dessa forma, podemos caracterizar a HPC Shelf como uma nuvem PaaS baseada em componentes para Computação de Alto Desempenho, que com nossa contribuição terá suporte para adaptação em tempo de execução. Ao permitir o processamento e resolução de contratos contextuais, a HPC Shelf oferece uma maneira uniforme para descrever as políticas e mecanismos de elasticidade disponíveis, permitindo uma ponte entre os seus atores, com diversos níveis de especialização.

O restante do capítulo encontra-se organizado da seguinte forma. As Seções 6.1 e 6.2 apresentam cenários para descrever situações nas quais adaptação é importante. A arquitetura do nosso *framework*, incluindo a organização dos componentes e os serviços necessários, está descrita na Seção 6.3. As conclusões são apresentadas na Seção 6.4.

6.1 Motivação

Nesta seção, delineamos cenários nos quais a reconfiguração elástica de aplicações de Computação de Alto Desempenho é importante. Na Seção 4.2, apresentamos vários projetos que comprovam a utilidade da reconfiguração elástica para Computação de Alto Desempenho, enquanto, na Seção 6.2, discutimos uma modelagem em alto nível de como esses sistemas podem ser estruturados de acordo com o tipo de aplicação.

Como discutimos na Seção 5.7, o usuário provedor espera que a execução das computações se comportem de acordo com os parâmetros de QoS descritos em contrato contextuais. Esses valores são determinados através de uma função de estimativa que analisa os contratos contextuais de plataformas e computações que executam sobre elas, sendo que essa função é definida pelo desenvolvedor de componentes. Por mais que o desenvolvedor tenha domínio das técnicas de desenvolvimento de código paralelo e da análise da complexidade dos algoritmos que implementa, o cenário em que a função de estimativa retorna valores inadequados não pode ser descartado. Tal situação pode ocorrer devido a limitações do conhecimento do desenvolvedor sobre as técnicas de análise ou devido ao cadastro de um novo perfil de plataforma na nuvem de serviços. Vimos, no Capítulo 2, que as infraestruturas para Computação de Alto Desempenho evoluem no decorrer do tempo. Portanto, uma estimativa estática baseada em *clusters* computacionais físicos pode não ser adequada para uma estrutura com maior distribuição, sejam grades ou nuvens computacionais.

Além de limitações em relação ao desenvolvedor de componentes, também devemos considerar incertezas produzidas pelo esforço do provedor de aplicações. É esse o ator que irá traduzir os requisitos em alto nível de usuários especialistas em valores para os pesos e prioridades atribuídos aos parâmetros de qualidade de serviço. A relação dos valores ocorre através da aplicação construída pelo provedor com o *framework* de alto

nível SAFe já existente. Entretanto, a aplicação pode interpretar a entrada de maneira equivocada, escolhendo componentes inadequados em relação aos requisitos impostos pelo usuário especialista, em nível mais alto de abstração em relação aos contratos contextuais.

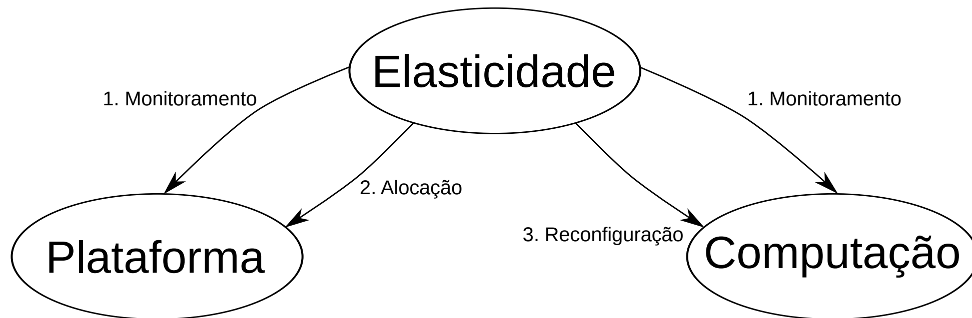
Ao disponibilizar um perfil de plataforma em uma nuvem de serviços HPC Shelf, o mantenedor garante a disponibilidade de alocação de recursos. Entretanto, se a infraestrutura de suporte for uma nuvem computacional IaaS, há possibilidade de interferência oriunda de outras aplicações em execução. Problemas decorrentes do aspecto *multi-inquilino* (*multi tenancy*) são de maior impacto para Computação de Alto Desempenho. Por exemplo, considerando que um servidor físico na infraestrutura que suporta a plataforma tem 12 núcleos de processamento e 24 GB de memória principal. Em teoria, duas máquinas virtuais cada uma com 6 núcleos e 12 GB de memória podem habitar o servidor, mesmo sendo cada máquina virtual (VM) de uma aplicação diferente. Componentes de Computação de Alto Desempenho são sensíveis ao uso compartilhado da hierarquia de memória. Existindo *cache* compartilhado entre os núcleos, instruções de uma máquina virtual podem inutilizar os valores carregados por outra VM. Dessa forma, mesmo com a divisão dos recursos, o desempenho é afetado. Interferência semelhante pode ocorrer no acesso à interface de rede. A mudança no conjunto de recursos alocados pode ajudar a sanar a interferência, seja instanciando novas máquinas virtuais em servidores com menor interferência ou eliminando a VM que está atrasando a computação como um todo.

6.2 Reconfiguração Elástica

De acordo com a classificação apresentada na Seção 2.3, podemos ter elasticidade em tempo de execução para aplicações paralelas maleáveis e evolutivas. O código maleável é reconfigurado de acordo com decisões tomadas por um agente externo. Na Figura 6.1, temos uma visão em alto nível do processo de reconfiguração para aplicações maleáveis. Um agente externo à computação, que pode executar um laço de decisão, monitora tanto a plataforma que fornece os recursos quanto a computação que executa nesses recursos. De acordo com as informações de estado coletadas, o agente de elasticidade realiza ações de alocação na plataforma e aplica as reconfigurações elásticas correspondentes na computação. O agente externo não precisa ser o mesmo para todos os tipos aplicações. Por exemplo, para aplicações altamente acopladas, abordagens semelhante ao Elastic MPI (ver Seção 4.2.3) são adequadas. Entretanto, a mesma política de decisão não é ideal para aplicações *bag of tasks*.

A *reconfiguração maleável* é interessante para permitir o controle de aplicações legadas ou no caso em que o desenvolvedor deseja reutilizar o agente de elasticidade para várias aplicações diferentes. Se o desenvolvedor deseja um controle maior da execução, a

Figura 6.1: Reconfiguração em código maleável.



Fonte: Elaborada pelo autor.

solução é o desenvolvimento de código evolutivo. Nesse tipo de aplicação, as chamadas para alteração dos recursos são feitas pela própria computação através de uma biblioteca padronizada. Como apresentado na Figura 6.2, o módulo de elasticidade apresenta essa biblioteca à computação e traduz as chamadas de acordo com os comandos necessários para alteração dos recursos na plataforma. Essa tradução é necessária porque os comandos de alocação para uma nuvem IaaS são diferentes das ações em um *cluster* físico. O módulo de elasticidade para código evolutivo não é um agente. Portanto, não há um laço de decisão em execução. A resposta de reconfiguração do módulo de elasticidade evolutivo são confirmações das requisições feitas pela computação. Essas respostas contém as informações sobre o recursos que a computação necessita para se ajustar. Por exemplo, no caso da requisição de novos nós de processamento, a resposta de reconfiguração contém os endereços IP das novas máquinas.

Figura 6.2: Reconfiguração em código evolutivo.

Fonte: Elaborada pelo autor.

Uma visão do conjunto de métodos que o módulo de elasticidade pode oferecer para aplicações evolutivas está na Tabela 6.1. Os métodos principais são aqueles que tratam da elasticidade vertical (adição de núcleos e memória) e da horizontal (adição de nós de processamento). Também reconhecemos a necessidade de métodos para a computação evolutiva inspecionar o estado da plataforma. O módulo elástico evolutivo também pode ser estendido pelo desenvolvedor. Por exemplo, métodos para controlar o consumo de

Tabela 6.1: Chamadas da biblioteca de elasticidade para aplicações evolutivas.

Método	**Descrição**
add_vcpu (int N)	adicionar núcleos aos nós de processamento
rem_vcpu (int N)	remover núcleos dos nós de processamento
add_node (int N)	adicionar nós de processamento
rem_node (int N)	remover nós de processamento
add_memory (int N)	adicionar N *megabytes* de memória aos nós
rem_memory (int N)	remove N *megabytes* de memória dos nós
get_free_mem()	retorna a quantidade de memória livre para alocação por nó
get_available_vcpus()	retorna a quantidade de núcleos para alocação por nó
get_free_nodes()	retorna a quantidade de nós

Fonte: Elaborada pelo autor.

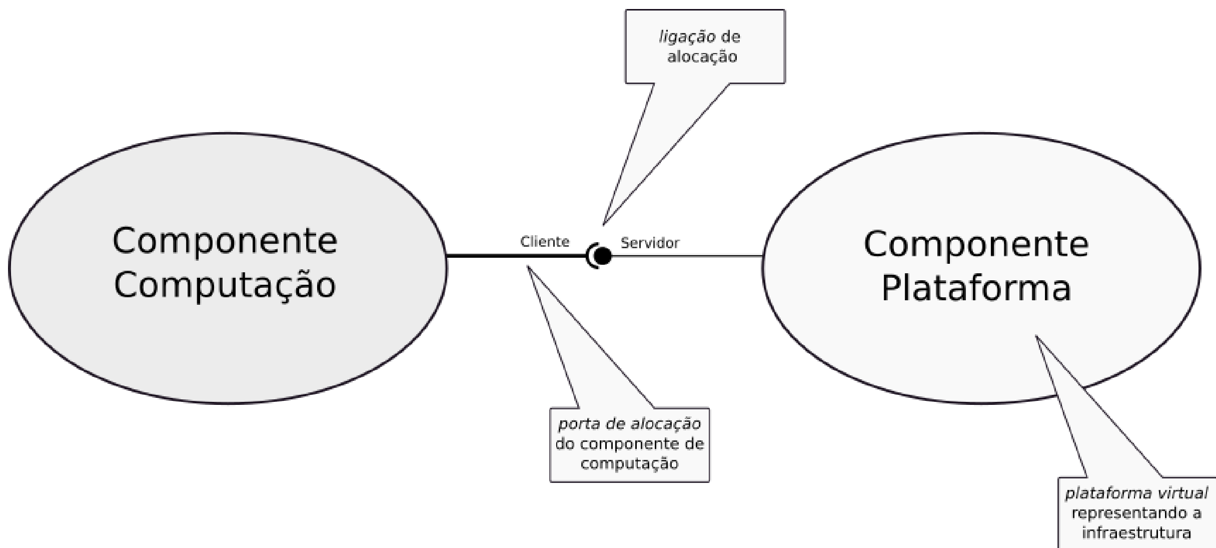
energia da computação (alteração do *clock* dos processadores) podem ser adicionados. Outro caso seria a remoção dos métodos de elasticidade vertical se esse tipo de reconfiguração não for suportado pela plataforma.

6.3 Arquitetura do Ambiente

Com os serviços e funcionalidades da HPC Shelf definidos no Capítulo 5, nesta seção descrevemos como ampliar esse ambiente para tratar os interesses de adaptação em tempo de execução apresentados na Seção 6.1. Nós definimos a arquitetura em termos dos componentes necessários e suas ligações. Importante lembrar que apesar de apresentarmos os componentes nas ilustrações como entidades disjuntas, seu modelo de implantação obedece o padrão SCMD (ver Seção 3.1.2) através da composição por sobreposição de unidades (ver Seção 5.2).

Um conceito importante da HPC Shelf que utilizamos é o *sistema computacional* (também conhecido como *componente de sistema*). Um sistema computacional é qualquer par formado por componente da espécie computação ligado a um componente da espécie plataforma (plataforma virtual) na arquitetura de um sistema de computação paralela. De fato, um sistema de computação paralela é formado por um conjunto de sistemas computacionais, possivelmente compartilhando plataformas virtuais. A ligação entre os dois componentes é feita através de uma porta provedora de alocação que a plataforma disponibiliza para a computação. Por sua vez, o componente de computação pode ser resultado da composição por sobreposição de outros componentes. A Figura 6.3 representa um componente de sistema, ou seja, a visão que o provedor de aplicações da HPC Shelf possui dos componentes que representam interesses da aplicação, implantados em uma

Figura 6.3: Ligação de componentes em um sistema computacional.



Fonte: Elaborada pelo autor.

infraestrutura de computação paralela através do Back-End. Para o provedor, o sistema computacional é um componente de primeira ordem, com suas próprias portas usuárias e provedoras.

A reconfiguração elástica de um sistema computacional é coordenada pelos parâmetros de qualidade nos contratos contextuais do componente de computação e no componente plataforma associado. De fato, o mecanismo de resolução de contratos sintetiza um contrato contextual para o sistema computacional, juntando-se esses dois contratos no mínimo contrato, em termos de restrições, compatível com ambos. Isso é possível pelo fato de que o contrato contextual (perfil) da plataforma virtual associada por meio da porta de alocação ser compatível, de acordo com o sistema de tipos, às restrições aos parâmetros de plataforma impostas no contrato do componente de computação, de forma que, no contrato contextual do sistema computacional, é suficiente aplicar os argumentos de contexto da plataforma virtual para os parâmetros de plataforma.

Vale ressaltar que, muito embora os argumentos aplicados aos parâmetros de qualidade em contratos contextuais sejam informados pelos provedores de aplicações, nada impede que possam ser informados, em mais alto nível, por usuários especialistas, sendo que a aplicação criada pelo provedor é responsável por traduzir os parâmetros de alto nível definidos no domínio do especialista para a representação do contrato contextual que é enviado ao Core para resolução. O sistema computacional instanciado na infraestrutura pelo Core através do serviço Back-End tem acesso ao contrato definido pelo algoritmo de resolução da HPC Shelf. Como o algoritmo garante compatibilidade entre a computação e a plataforma, o contrato do componente de sistema atualiza os argumentos de contexto relativos à plataforma presentes no contrato da computação pelos valores presentes do

contrato da plataforma, obedecendo a relação de tipos. Portanto, os requisitos de qualidade de serviço são acessíveis pelo componente de sistema instanciado. Um exemplo de um contrato contextual para um sistema computacional:

ComputationalSystemMatMulCluster [
  computation = MatrixMultiplicationMPI [***matrix_type*** = SquareMatrix, ***sparsity*** = 0.1]
  platform = AcademicCluster [
    node_range = 2:10,
    processor = Xeon5640 [***core_range*** = 1:8]
    memory = Four8GbSticksMemoryCluster [***quantity_range*** = 2:24]]
  qos_contract = [
    qos_values = {0.9, 0.7, 1500, 15, 5},
    qos_weigths = {0.7, 0.3, 0.8, 0.1, 0.1},
    monitor_interval = 100,
    reconfiguration_interval = 600,
    sample_interval = 300,
    alfa = 0.1]]

Com dito anteriormente, para descrever o contrato do componente de sistema, precisamos de parâmetros e argumentos dos contratos da computação e da plataforma. No caso da computação (***computation***), para o suporte da reconfiguração elástica, não há alterações. Na descrição da plataforma (***platform***), temos uma alteração em relação ao discutido na Seção 5.5. No lugar de valores fixos para número de nós de processamento, quantidade de núcleos e tamanho da memória por nó, temos agora parâmetros *qualificadores* que representam as faixas de valores possíveis (***node_range***, ***core_range*** e ***quantity_range***). Esses valores correspondem aos limites que o mantenedor da plataforma garante para o perfil escolhido para o componente de sistema. Outra modificação é a inclusão do contrato de QoS. Temos os valores estimados para as métricas de QoS (acurácia, eficiência, tempo de execução, custo e consumo energético) e os pesos designados para cada característica. Os últimos parâmetros do contrato são importantes para o caso maleável e são explicados nas seções seguintes. Para o contrato ampliado com parâmetros qualificadores, damos o nome de *contratos contextuais elásticos*.

O contrato do componente de sistema, após ser formado pelo processo de resolução e classificação, é encaminhado junto com os componentes para o serviço Back-End. Ele é utilizado para coordenar o interesse da adaptação. Propomos a definição de um componente aninhado chamado *componente de reconfiguração*. Esse componente terá acesso ao contrato e será responsável por interagir com os outros integrantes do sistema computacional para cumprir os requisitos de qualidade de serviço definidos no contrato contextual através de ações de reconfiguração. A localização desse componente dentro do sistema depende do tipo de reconfiguração adotado, como veremos a seguir.

6.3.1 Reconfiguração Elástica de Componentes na HPC Shelf

Para provedores de aplicações e usuários especialistas, interessa que o sistema computacional se adapte em busca da manutenção dos parâmetros de qualidade de serviço descritos no contrato. Para isso, fazemos a distinção entre componentes de sistema *não reconfiguráveis* e *reconfiguráveis*. Os primeiros já são suportados pelas espécies atuais da HPC Shelf (ver Seção 5.3), notadamente através dos componentes de computação e plataformas virtuais, podendo ser *rígidos* ou *moldáveis*, segundo a classificação de tarefas paralelas discutida na Seção 2.3. Para o *framework* proposto nesta Tese, incluímos os últimos, tornando possível sistemas computacionais *maleáveis* e *evolutivos*, a fim de lidar com os interesses da reconfiguração elástica.

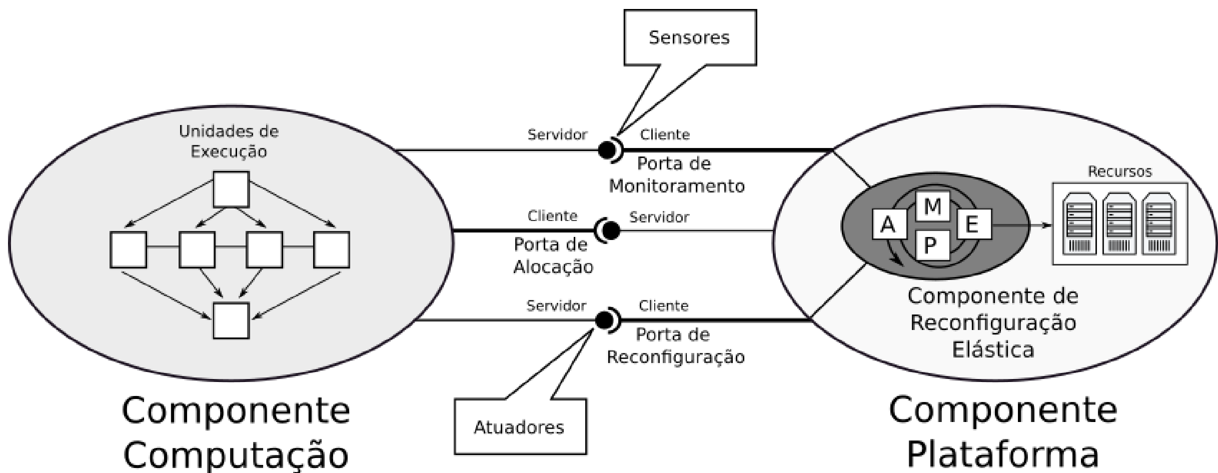
Além da divisão entre os dois cenários, estabelecemos que os componentes aninhados de um sistema computacional possuem variáveis *sensores* e parâmetros *atuadores* em suas portas de ambiente. As variáveis sensores caracterizam o estado interno do componente, expondo apenas as informações que são de interesse do processo de reconfiguração. Atuadores são parâmetros de configuração que podem alterar o comportamento do componente. Essa definição é uma generalização dos conceitos utilizados em (MCINNES, 2006b). Dependendo do caráter maleável ou evolutivo do sistema computacional, tanto o componente computação quanto o de plataforma podem apresentar sensores e atuadores.

6.3.1.1 Sistemas Maleáveis

Considerando a reconfiguração maleável, a ligação dos componentes de computação e plataforma é mostrada na Figura 6.4. Um componente de sistema maleável permite ajustar o conjunto de recursos durante a execução, seja através da elasticidade horizontal ou vertical. Entre os componentes aninhados de computação e plataforma, existem três portas de ambiente: *alocação*, *monitoramento* e *reconfiguração*. A porta de alocação é uma porta de ambiente provedora da plataforma, na qual a computação se conecta na implantação do componente de sistema pelo Back-End. No momento da ativação dessa porta, a computação recupera as informações necessárias para acessar o conjunto inicial de recursos. Por padrão, o conjunto inicial é configurado com os valores mínimos permitidos pelos parâmetros qualificadores no contrato contextual da plataforma. Além da troca inicial de informações, ao desativar essa porta, o componente de computação sinaliza à plataforma virtual o término da execução.

A computação fornece, no papel de servidor, duas portas relacionadas com a reconfiguração elástica. A porta de monitoramento fornece informações sobre o estado da computação no formato de valores para variáveis sensores. Há uma variável sensor para a acurácia, que retorna a precisão do resultado obtido até o momento de sua leitura. Para

Figura 6.4: Ligação de componentes em um sistema computacional maleável.



Fonte: Elaborada pelo autor.

muitos casos, o valor não tem significado quando a computação implementa algoritmos numéricos exatos. Entretanto, para métodos aproximativos como Monte Carlo, o desenvolvedor de componentes cria mecanismos para atualizar essa variável. Adotamos como faixa de valores possíveis o intervalo $[0.0, ..., 1.0]$, que representa o percentual de precisão em relação ao esperado. Outra variável sensor presente mede o *progresso* da computação. Ela é atualizada a cada etapa que o algoritmo completa. Por exemplo, se a computação precisa analisar N itens de dados, a cada item processado, a variável progresso é atualizada para o número de itens já analisados sobre o total N. Qualquer computação com um laço externo principal pode utilizar a variável índice para calcular o valor do progresso. Assim como a variável da acurácia, o progresso apresenta valores na faixa $[0.0, ..., 1.0]$.

Listagem 6.1: Laço MAPE para Componentes de Sistema Maleáveis.

```
1  inicio = plataforma.lerData()
2  ultima_reconfiguracao = inicio
3  while porta_alocacao.finalizado() != True :
4      bloquear(contrato.monitor_interval)
5      # Monitorar
6      (acuracia, progresso) = porta_monitoramento.lerSensores()
7      eficiencia = plataforma.lerSensores()
8      # Analisar
9      (tempo, custo, consumo) = previsao(historico, progresso)
10     amostra = { "acuracia" : acuracia, "eficiencia" : eficiencia,
11                 "tempo" : tempo, "custo" : custo, "consumo" : consumo}
12     tempo_atual = plataforma.lerData()
13     recursos_atuais = plataforma.lerRecursos()
14     historico[tempo_atual] = ("amostra", amostra, progresso, recursos_atuais)
15     # Planejar
16     quebra_de_contrato = False
17     alfa = contrato.alfa()
```

```
18        for p in ["acuracia", "eficiencia", "tempo", "custo", "consumo"]:
19            if not ((amostra[p] > (1 − alfa) ∗ contrato.valor[p])
20                    and (amostra[p] < (1 + alfa) ∗ contrato.valor[p])) :
21                delta[p] = abs(contrato.valor[p] − amostra[p])
22                delta[p] = delta[p] / contrato.valor[p]
23                delta[p] = delta[p] ∗ contrato.peso[p]
24                quebra_de_contrato = True
25        if not quebra_de_contrato :
26            continue
27        parametro = maximo_por_valor(delta)
28        acao = reconfigurar_parametro(recursos_atuais, historico, parametro)
29        # Executar
30        atual = plataform.lerData()
31        if (atual − ultima_reconfiguracao) < contrato.reconfiguration_interval:
32            continue
33        ultima_reconfiguracao = atual
34        novos_recursos = plataforma.alocarRecursos(acao)
35        porta_reconfiguracao.atuador(novos_recursos)
36        tempo_reconfiguracao = plataforma.lerData()
37        historico[tempo_reconfiguracao] = ("reconfiguracao", acao,
38                                          recursos_atuais, novos_recursos)
39        contrato.atualizarAlfa(historico)
```

A porta de reconfiguração da computação possui um método atuador que recebe um novo conjunto de recursos da plataforma toda vez que a reconfiguração é realizada. O código do componente de computação, ao receber os novos recursos, adapta sua execução para o novo cenário. Essa adaptação envolve reajuste nas unidades instanciadas, criando ou eliminando processos e *threads*. O desenvolvedor de componentes deve implementar o componente de computação maleável com suporte a ações de expansão/redução e redimensionamento. O primeiro tipo de ação representa a elasticidade horizontal, ou seja, permite à plataforma controlar quantos nós de processamento a computação executa sobre. O segundo tipo de ação representa a elasticidade vertical. Novamente, a plataforma só informará recursos que estão dentro dos limites do parâmetros qualificadores.

Enquanto a computação fornece informações atuais de estado através sensores e recebe instruções de reconfiguração com ações de elasticidade através do atuador, afirmamos que um componente plataforma é maleável se possui aninhado um componente de reconfiguração. Um laço MAPE (Seção 4.1.3.1) executa interno ao componente. A Listagem 6.1 apresenta a estrutura geral desse laço. O contrato do componente de sistema está disponível para o laço desde o início da execução. Antes das iterações, o momento que marca o início do sistema é armazenado na variável *inicio*, que é copiado para a variável `ultima_reconfiguracao`. Sempre que ocorrer reconfiguração do sistema, o horário será atualizado nessa variável.

O laço executa de maneira ininterrupta até o componente de computação finalizar sua atividade. Entretanto, a primeira ação (linha 4) é bloquear o processo durante um intervalo definido em segundos no contrato (`monitor_interval`). A razão desse bloqueio é evitar que o processo executando o laço se torne oneroso ao mesmo tempo em que o intervalo em espera permite que a aplicação execute tempo suficiente para indicar se uma ação de reconfiguração teve efeito ou não. Esse intervalo pode ser configurado pelo desenvolvedor de componentes no contrato. Por exemplo, se um componente executa por horas, não é indicado monitorá-la a cada 5 segundos. Um intervalo maior será mais representativo do verdadeiro estado do sistema.

Após o fim do bloqueio, temos a **fase de monitoramento** do laço (linhas 6 e 7). O estado do componente de computação é recuperado nas variáveis `acuracia` e `progresso`, através da porta de monitoramento. Já o valor de `eficiencia` é calculado a partir da carga do sistema. Realizando a divisão da carga do sistema pelo número de núcleos disponíveis na plataforma, temos a eficiência do sistema.

A **fase de análise** (linhas 9 a 14) define uma estimativa para o tempo total de execução, custo e consumo energético a partir do histórico da execução do sistema e do progresso atual. A função `previsao` é definida pelo desenvolvedor de componentes, de acordo com a natureza da computação. No caso de algoritmos uniformes, que o tempo de execução cresce de maneira linear com o progresso, uma função de extrapolação linear pode ser aplicada. Entretanto, computações com comportamento de maior variabilidade exigem funções de análise mais refinadas. Os valores de todos os parâmetros são consolidados em uma estrutura chamada `amostra`, que é armazenada junto com o valor de `progresso` e os recursos atuais no histórico da execução, indexados pelo momento da coleta.

Após coletar e consolidar informações sobre o estado da computação e da plataforma, a **fase de planejamento** (Linhas 16 a 28) decide qual a ação de reconfiguração a ser tomada. Para tal, os pesos fornecidos pelo provedor de aplicações no contrato são utilizados para ordenar os parâmetros que justificam a reconfiguração. Para cada parâmetro, o laço verifica se os valores da amostra estão dentro dos limiares definidos pelos valores do contrato e a variável `alfa`, também definida pelo provedor de aplicações. Através de `alfa`, o provedor pode definir a tolerância ao cumprimento do contrato. Se todos os parâmetros na amostra estiverem dentro da faixa de tolerância, então o laço pula para a próxima iteração sem nenhuma ação. Caso contrário, para os parâmetros em quebra de contrato, o valor `delta` é calculado como a diferença entre o valor esperado e o encontrado na amostra, normalizado pelo valor esperado e em seguida multiplicado pelo peso. O parâmetro que apresentar o maior valor `delta` é aquele que está em quebra de contrato e representa o maior interesse do provedor. Portanto, é o parâmetro escolhido para guiar a reconfiguração.

Ainda na fase de planejamento, o parâmetro com maior `delta` é informado à função `reconfigurar_parametro` junto com o histórico da execução e os recursos atuais. Essa função é definida pelo desenvolvedor de componentes e deve retornar qual ação de elasticidade deve ser realizada. Para exemplificar, considere o parâmetro de desempenho. Em um dado momento, o tempo de execução está acima do valor esperado. A solução seria adicionar mais recursos na tentativa que mais núcleos ou nós de processamento consigam finalizar o cálculo mais rápido. Entretanto, caso o desenvolvedor tenha feito uma análise de isoeficiência (ver Seção 2.4), é possível determinar se a computação já atingiu o limite de escalabilidade. Nesse caso, aumentar o conjunto de recursos não terá, teoricamente, o efeito esperado.

A ação de elasticidade adotada pode ser do tipo vertical ou horizontal. Apesar de ficar a cargo da função `reconfigurar_parametro` decidir qual a alteração a ser feita no recurso, há consenso na área que no lugar de grandes mudanças no conjunto de recursos, o ideal é realizar pequenas alterações e avaliar seu impacto para só então decidir prosseguir no mesmo sentido (ROY; DUBEY; GOKHALE, 2011). Uma razão para essa estratégia é que adicionar um núcleo ou um nó de processamento consome menos tempo do que adicionar vários núcleos ou máquinas. Se a função realizar uma estimativa acima do ideal, será gasto muito tempo na reconfiguração, o que resultará em desperdício de recursos. Adicionando recursos através de granularidade fina, o laço pode avaliar se cada ação teve sucesso ou não, apenas prosseguindo em uma direção que tem maior chance de retorno comprovado. Pelo mesmo raciocínio, para as plataformas que a suportam, é melhor exaurir a elasticidade vertical antes de partir para a aplicação de ações horizontais, visto que adicionar ou remover núcleos é menos oneroso que adicionar ou remover nós de processamento.

Finalizando o laço MAPE, temos a **fase de execução** (Linhas 30 a 39). A primeira verificação é garantir um intervalo de tempo mínimo entre duas reconfigurações, definido no contrato. Em seguida, o momento da última reconfiguração é atualizado, o novo conjunto de recursos é definido na plataforma de acordo com a ação de elasticidade e a computação é informada da adaptação. Apesar de não estar explícito no código, existem duas situações com comportamentos diferentes. No caso do aumento de recursos, a plataforma pode realizar a alocação e só depois informar a computação, com menor sobrecarga de sincronização. Entretanto, se for a remoção ou diminuição de recursos, a computação e a plataforma são sincronizadas de forma que não seja eliminado o elemento no qual ainda está em execução um processo ou *thread*. Após a reconfiguração ser concluída com sucesso, o histórico é atualizado com a ação que foi feita, os recursos anteriores e o novo conjunto de recursos. A última tarefa do laço é invocar uma função chamada `atualizarAlfa` definida pelo provedor de aplicações. Essa função analisa o histórico e flexibiliza ou restringe os limiares dos parâmetros. Caso deseje manter um valor fixo, o provedor pode retornar o mesmo valor inicial do contrato.

Tabela 6.2: Funções para especialização do laço MAPE.

Função	**Argumentos**	**Objetivo**	**Responsável**
`previsao`	histórico, progresso.	Estimar valores dos parâmetros.	Desenvolvedor de Componentes.
`reconfigurar_parametro`	recursos atuais, histórico, parâmetro.	Definir ação de elasticidade.	Desenvolvedor de Componentes.
`atualizarAlfa`	histórico	Atualizar limiares.	Provedor de Aplicações.

Fonte: Elaborada pelo autor.

Na Tabela 6.2, resumimos as funções que permitem a especialização do laço MAPE. Para tornar a discussão de sistemas maleáveis completa, precisamos apena esclarecer qual seria o formato da ação de elasticidade. A ação é uma estrutura que define valores para os parâmetros da plataforma que representam a granularidade dos recursos. Em suma, uma ação tem um valor para quantidade de nós, número de núcleos por nós e total de memória por nó. Para realizar elasticidade horizontal, a ação deve alterar a quantidade de nós em relação os recursos atuais, que são passados como argumentos para `reconfigurar_parametro`. A elasticidade vertical é realizada para alterar os outros valores, tais como quantidade de núcleos e memória por nó. Ao receber uma ação, a função de alocação da plataforma trata de realizar as alterações necessárias.

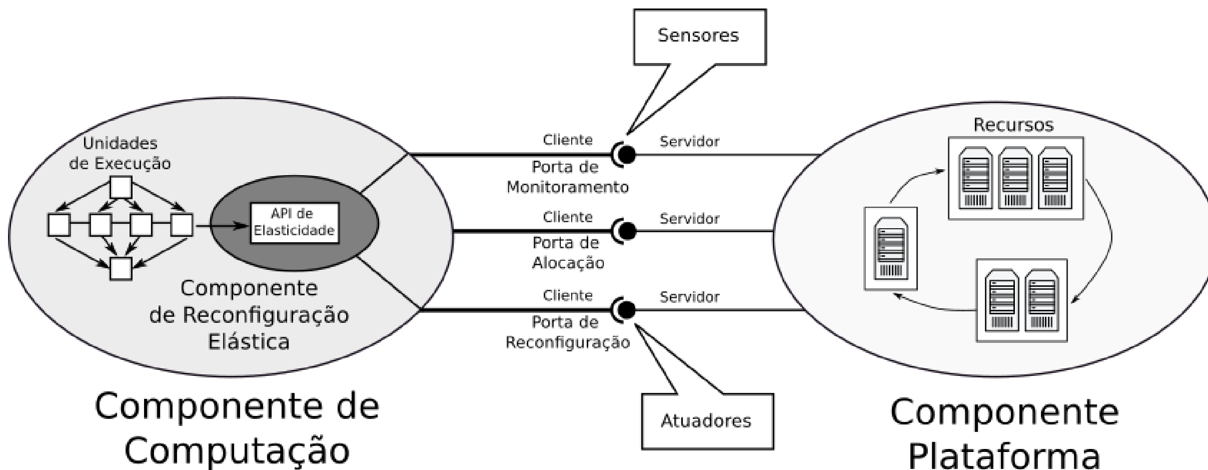
6.3.1.2 Sistemas Evolutivos

Sistemas computacionais evolutivos transferem as decisões sobre reconfigurações da plataforma para a computação. A plataforma torna-se provedora da porta de reconfiguração que o componente de computação invoca para requisitar ajustes nos recursos. A Figura 6.5 apresenta o cenário evolutivo. As variáveis sensores na porta de monitoramento agora representam o estado da plataforma, que incluem a eficiência, custo e consumo energético. Entretanto, nada impede que o componente de computação ignore as variáveis sensores da plataforma e determine reconfigurações baseadas apenas em sua lógica interna. Portanto, no contexto da reconfiguração evolutiva, um componente de computação é dito evolutivo se possui a ele aninhado um componente de reconfiguração.

A porta de reconfiguração da plataforma apresenta métodos atuadores para alteração do conjunto de recursos, assim como descritos na Tabela 6.1. O componente de reconfiguração interno à computação analisa o recursos atuais, estabelece um novo conjunto de recursos objetivo e utiliza os métodos atuadores para submeter à plataforma as requisições. Apesar de não executar mais o laço de decisão, a plataforma ainda verifica se as invocações da computação na API de elasticidade não infringem os limites definidos no

contrato do sistema. Apenas se as alterações forem aceitas é que a computação recebe a confirmação dos novos recursos.

Figura 6.5: Ligação de componentes em um sistema computacional evolutivo.



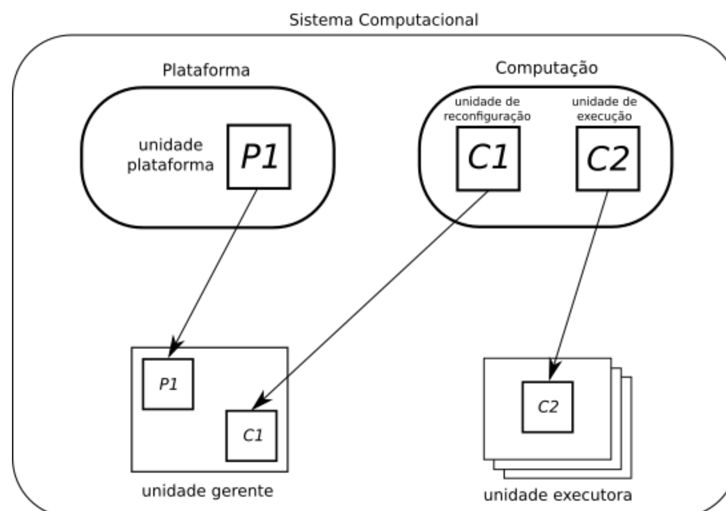
Fonte: elaborada pelo autor.

É papel do desenvolvedor de componentes implementar o componente de reconfiguração. No cenário evolutivo, esse ator tem papel predominante no controle da reconfiguração. No caso maleável, ele fornece funções que são invocadas no momento da reconfiguração, mas ele não tem controle completo de quando a adaptação deve ocorrer. Consideramos o caso evolutivo adequado para computações que adotam *decomposição exploratória* ou *decomposição especulativa* (KUMAR, 1994). São problemas que consistem em buscas em um espaço de possíveis soluções, sendo que em determinada etapa do algoritmo é feita uma opção por qual direção nesse espaço é a mais indicada de fornecer uma solução ideal. Portanto, a definição do melhor conjunto de recursos para o restante da execução não depende do histórico da execução, mas sim da lógica interna do componente de computação.

6.3.2 Estrutura Interna de Componentes Paralelos Elásticos

Explicamos como um componente de computação maleável ou evolutivo interage com o resto do sistema para elaborar a elasticidade. Entretanto, para o componente ter suporte às interações detalhadas para cada caso, ele precisa ser desenvolvido considerando sua reconfiguração interna. Na HPC Shelf, adotamos o modelo Hash. Dessa forma, a reconfiguração interna de um componente implica na realocação das suas unidades paralelas. Na Figura 6.6, apresentamos um exemplo da estrutura interna de um componente de sistema elástico. Representamos componentes aninhados de dados e computação, sendo a representação válida para o caso evolutivo e maleável.

Figura 6.6: Estrutura interna de um componente Hash elástico.



Fonte: Elaborada pelo autor.

A *unidade gerente* do componente de sistema é formada por uma unidade do componente plataforma e da computação. No caso maleável, a unidade da plataforma será a representação do laço MAPE, enquanto na unidade da computação temos as funções implementadas pelo desenvolvedor, fornecendo acesso às variáveis sensores da porta de monitoramento e ao método atuador da porta de reconfiguração. No caso evolutivo, a unidade da plataforma representa a API elástica e os sensores e atuadores da plataforma, enquanto a unidade da computação encapsula a lógica de reconfiguração do componente.

Em ambos os tipos de sistema, no componente aninhado temos a *unidade de execução* que representa o algoritmo paralelo implementado. Essa unidade é mapeada para uma unidade paralela do componente de sistema, sendo que durante a execução a reconfiguração altera a quantidade de processos instanciados que executam essa unidade.

Apresentamos a reconfiguração como uma maneira de habilitar a elasticidade na HPC Shelf. A adaptação é interna aos componentes, sendo que suas unidades são reestruturadas para o novo conjunto de recursos computacionais. Além da reconfiguração elástica, um sistema computacional também pode suportar a reconfiguração arquitetural, de maneira semelhante como apresentado na Seção 3.1.3. Esse tipo de adaptação envolve alterações estruturais, entre as quais a substituição de componentes e o estabelecimento ou remoção de ligações. Esse tipo de reconfiguração abre muitas possibilidades de adaptação, mas impõe desafios em relação a manutenção dos parâmetros de qualidade definidos no contrato contextual do sistema. Nesta tese, decidimos tratar a reconfiguração elástica e incluir a reconfiguração arquitetural nos planos para o futuro da HPC Shelf.

6.4 Conclusões

Para o pesquisador que utiliza computação paralela na simulação de fenômenos científicos, a diversidade de plataformas paralelas é ao mesmo tempo benéfica, pois representa mais ciclos computacionais para resolução de problemas, e problemática, já que o pesquisador precisa monitorar a execução e tomar ações para garantir o uso adequado dos recursos computacionais. O *framework* proposto neste capítulo permite que a gerência seja automatizada através da definição de componentes que controlam os interesses de reconfiguração.

Um dos objetivos das nuvens computacionais é a virtualização de recursos, permitindo controlá-los de maneira programática. Através do tratamento do interesse reconfiguração elástica, apresentamos uma contribuição que avança o estado da arte de acordo com a metodologia das nuvens, sendo capaz de aprimorar a execução de aplicações de Computação de Alto Desempenho nas infraestruturas modernas.

Neste capítulo, apresentamos os cenários motivadores para nosso *framework* e uma visão arquitetural de como aprimoramos o projeto da nuvem HPC Shelf com a reconfiguração elástica de componentes. Dessa forma, defendemos que a HPC Shelf permite aos seus usuários tirar o melhor proveito das infraestruturas que têm a disposição, fornecendo adaptação em tempo de execução à possíveis mudanças de contexto. Precisamos agora apresentar estudos de casos que evidenciem a viabilidade da solução.

7. Estudos de Casos

Neste capítulo, apresentamos estudos de caso para o arcabouço de reconfiguração elástica de componentes paralelos descrito no Capítulo 6. Na Seção 7.1, realizamos a descrição do perfil de plataforma adotado nos experimentos. Em seguida, na Seção 7.2, apresentamos o comportamento de um ambiente de execução sem reconfiguração em comparação com um componente de sistema maleável. O objetivo é comprovar a eficácia do *framework* em guiar a execução de acordo com o contrato de QoS. Na Seção 7.3, apresentamos componentes de sistemas para o caso evolutivo. Por fim, na Seção 7.4, apresentamos as conclusões obtidas pelos experimentos.

7.1 Ambiente de Testes

Como infraestrutura paralela de suporte para realização da nossa avaliação, adotamos uma nuvem IaaS construída com a solução OpenStack. Tal *cluster* é composto por 6 servidores físicos como nós de processamento. Cada servidor tem dois processadores *Intel Xeon CPU E5-2650 v3* funcionando na frequência de 2.30 GHz, totalizando 20 núcleos de processamento por nó. A memória disponível em cada nó é de 64 GB. A rede de interconexão é da tecnologia *Gigabit Ethernet*. A versão do OpenStack utilizada é a *Liberty*, com a tecnologia de virtualização KVM. Durante a criação de máquinas virtuais, o sistema as aloca em servidores físicos do *cluster* usando distribuição *round robin*.

A partir da nuvem disponível, definimos a plataforma como um *cluster* virtual. A virtualização em KVM não suporta elasticidade vertical, portanto não exploramos essa característica nos nossos experimentos. Como discutido na Seção 6.3.1.1, para controlar a granularidade da plataforma, adotamos um modelo de máquina virtual com 2 núcleos de processamento e 4 GB de RAM, executando o sistema operacional Ubuntu Server versão *16.04.02 LTS*. Nós desenvolvemos um protótipo para o serviço Back-End utilizando a linguagem de programação Python versão 2.7.13. Os componentes de computação foram desenvolvidos utilizando os pacotes *mpi4py* e *numpy* executando sobre a versão 1.10.2 da distribuição MPI *OpenMPI*. Para o contrato do perfil de plataforma, além dos argumentos que descrevem o *hardware* da infraestrutura, a informação importante é o valor fornecido para o parâmetro ***node_range***. Definimos que a quantidade inicial de nós de processamento são 2 máquinas virtuais, sendo que através da reconfiguração a elasticidade horizontal permite alcançar até 10 VMs (VM, do inglês *Virtual Machine*).

7.2 Sistemas Computacionais Maleáveis

Visando avaliar o comportamento de sistemas computacionais maleáveis, desenvolvemos um componente de computação que realiza multiplicação de matrizes quadradas. A entrada fornecida é uma lista de matrizes. Por exemplo, considere a lista a seguir:

$$(1000, MA0, MB0)$$
$$(2000, MA1, MB1)$$
$$(1000, MA2, MB2)$$

O exemplo de entrada acima representa três multiplicações de matrizes. A primeira multiplicação é de duas matrizes 1000×1000, a segunda tem ordem de 2000×2000 e a terceira 1000×1000. Além das portas de reconfiguração e monitoramento, o componente tem uma porta `Matrix_Input` que recebe a lista de matrizes para multiplicação e retorna uma lista de arquivos com os resultados. Para o exemplo acima, a saída é no formato: $(MC0, MC1, MC2)$.

7.2.1 Comportamento do Sistema sem Reconfiguração

Para compreender o estado inicial do componente de sistema, submetemos listas de 20 matrizes de ponto flutuante densas como entrada para o componente de computação. Variamos o número de nós de processamento virtuais e o tamanho das matrizes. O resultado pode ser visto na Tabela 7.1. Medimos os valores para os parâmetros de desempenho (tempo de execução), eficiência (carga) e custo. Não calculamos valores para a acurácia pois a multiplicação de matrizes não é um algoritmo aproximativo. A medição do consumo energético com precisão depende de equipamentos não disponíveis no momento de escrita da Tese.

O tempo de execução apresentado para análise de desempenho é medido em segundos desde a criação do componente de sistema até sua finalização. A carga é definida como a razão entre a soma da carga de todas as máquinas virtuais sobre o número de núcleos de CPU. O custo é calculado em função dos número de máquinas virtuais alocadas, como demostrado na equação abaixo:

$$custo = \sum_{i=min_node}^{max_node} \sum_{j=min_core}^{max_core} i * c(j) * t(i,j), \tag{7.1}$$

Tabela 7.1: Comportamento do Sistema Computacional sem Reconfiguração.

VMs	Tamanho da Matriz	Tempo de Execução	Carga	Custo
2	1000	21,35	0,56	42,69
2	2000	105,41	1,05	210,83
2	3000	290,78	1,42	581,56
2	4000	652,99	1,61	1.305,98
2	5000	1.132,11	1,75	2.264,22
2	6000	1.962,21	1,81	3.924,42
2	7000	2.914,36	1,86	5.828,73
2	8000	4.254,82	1,93	8.509,64
2	9000	5.525,48	1,92	11.050,97
2	10000	7.565,08	1,93	15.130,17
6	1000	32,34	0,64	194,03
6	2000	96,71	1,05	580,25
6	3000	217,44	1,45	1.304,66
6	4000	424,55	1,47	2.547,32
6	5000	671,66	1,37	4.029,95
6	6000	1.028,39	1,46	6.170,32
6	7000	1.528,05	1,50	9.168,30
6	8000	2.091,99	1,52	12.551,95
6	9000	2.886,48	1,58	17.318,86
6	10000	3.637,13	1,61	21.822,80
8	1000	65,06	1,11	520,47
8	2000	118,16	1,41	945,28
8	3000	253,15	1,64	2.025,22
8	4000	415,64	1,58	3.325,10
8	5000	717,89	1,45	5.743,13
8	6000	993,45	1,53	7.947,62
8	7000	1.369,70	1,36	10.957,57
8	8000	1.911,45	1,53	15.291,62
8	9000	2.506,51	1,53	20.052,07
8	10000	3.155,27	1,58	25.242,17

Fonte: Elaborada pelo autor.

O termo $c(j)$ é o custo de um nó com j núcleos por unidade de tempo e $t(i, j)$ é o intervalo de tempo em que o conjunto de recursos foi composto por i máquinas virtuais com j núcleos de CPU. No nosso caso, $j = 2$, atribuímos $c(2) = 1$ e o valor de i no decorrer no ciclo de vida do componente de sistema é recuperado a partir do histórico da execução.

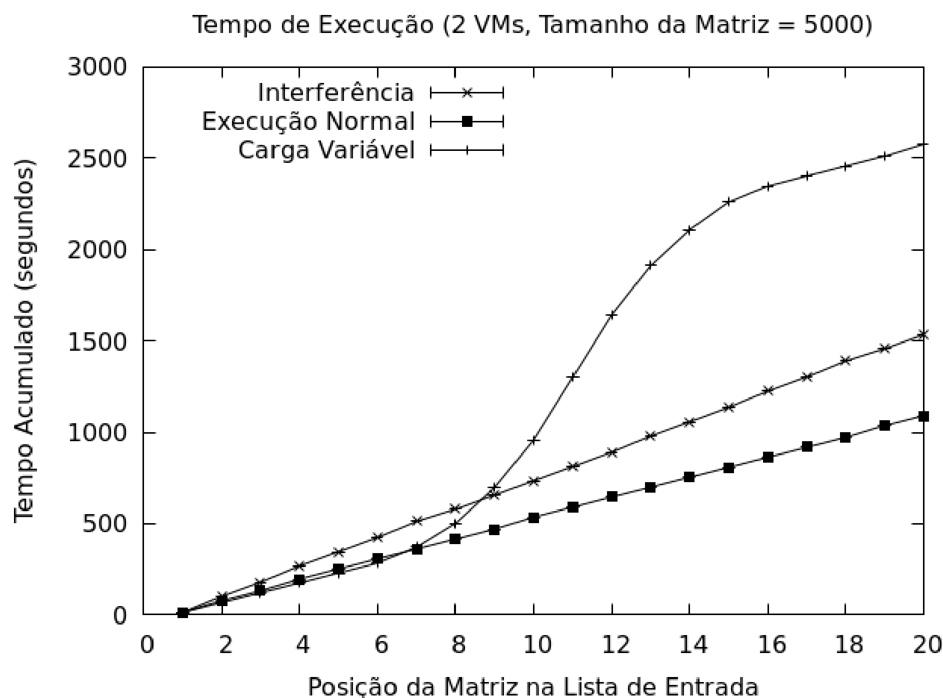
O que a Tabela 7.1 nos permite observar é que de 2 máquinas virtuais para 6, há um salto considerável de desempenho para valores altos de tamanho de matriz. Por exemplo, para matrizes 10000×10000, saímos de $7.565, 8$ segundos com 2 VMs para $3.637, 13$ segundos com 6 VMs. Entretanto, com 8 VMs, o valor só diminui para $3.155, 27$. O custo da execução, como é proporcional ao tempo de execução e não há variação no conjunto de recursos, se comporta de maneira similar. Como era esperado devido aos valores do tempo de execução, a carga sai de $1, 93$ com 2 VMs, valor que beira à sobrecarga do sistema, para $1, 61$ com 6 VMs e finalmente $1, 58$ com 8 máquinas virtuais. O que esses valores indicam é que a adição indiscriminada de recursos não necessariamente acarreta a melhoria de todos os parâmetros.

Para obtenção dos dados da Tabela 7.1, foram submetidas listas de matrizes com todas as entradas da mesma ordem e o sistema executou de forma exclusiva na plataforma. Porém, precisamos compreender o comportamento do ambiente quando o sistema é mal definido e há interferência na infraestrutura de nuvem, como discutido na Seção 6.1. Para tal, avaliamos a execução quando a carga submetida na lista de matrizes não é homogênea e com interferência oriunda da execução de outras máquinas na mesma infraestrutura.

As figuras 7.1 e 7.2 apresentam o comportamento do tempo de execução e eficiência para a execução bem definida exclusiva, com carga variável e interferência. A execução bem definida normal consiste de uma lista de 20 matrizes de ordem 5000. O contrato mal definido que implica na carga variável é simulado através da definição de uma lista de 20 matrizes na qual as 5 primeiras e as 5 últimas multiplicações da lista envolvem matrizes de ordem 5000. Da sexta até a décima posição na lista, a ordem das matrizes cresce de 6000 até 10000, e da décima primeira posição até a décima quinta a ordem decresce até o valor 6000 novamente. Em outras palavras, o valor da ordem é incrementado de 1000 em 1000 até 10000, depois decrementado na mesma proporção.

De acordo com a técnica de alocação *round robin*, ao criar uma plataforma virtual com duas máquinas virtuais, elas são alocadas em servidores físicos diferentes. A interferência foi simulada escolhendo um desses servidores para hospedar máquinas virtuais executando outra carga de trabalho. Durante a execução de um componente de sistema com uma lista homogênea de 20 matrizes de ordem 5000 nas duas VMs da plataforma, criamos 10 máquinas virtuais do mesmo perfil no servidor escolhido. Essas máquinas executaram o mesmo componente, mas com uma lista de 100 matrizes de ordem 10000. O objetivo de

Figura 7.1: Tempo de Execução sem Reconfiguração.



Fonte: Elaborada pelo autor.

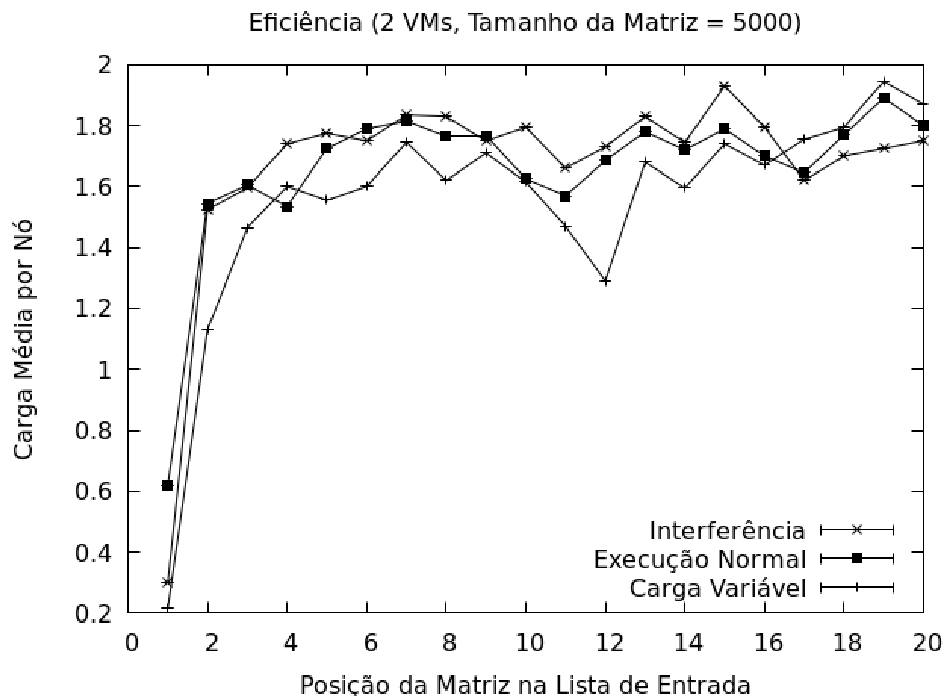
submeter uma entrada maior para o sistema causador da interferência foi garantir que a mesma ocorresse durante toda a execução do sistema em observação.

Na Figura 7.1, observamos que a entrada mal definida tem maior impacto no tempo de execução do que a interferência, sendo que ambos os cenários apresentam desempenho pior do que a execução normal. A eficiência do sistema não é alterada de forma significativa pelos cenários avaliados, como pode ser visto na Figura 7.2. Isto ocorre porque a entrada com matrizes de ordem 5000 já é suficiente para ocupar os recursos alocados e no caso da interferência a carga da máquina virtual afetada é transferida para a outra. No cálculo da média, o desequilíbrio converge. Como o conjunto de recursos não se altera durante a execução, podemos afirmar que o custo da execução em ambos os casos é maior, pois as mesmas máquinas virtuais são alocadas por um intervalo de tempo maior que a execução normal. No caso da execução normal, o custo, em valores inteiros, de 2264 unidades monetárias. Já o cenário da carga variável tem custo de de 5150, enquanto o cenário com interferência apresenta 3069 como orçamento.

7.2.2 Comportamento do Sistema com Reconfiguração

Para demonstrar a habilidade de reconfiguração dos componentes de sistema, definimos um contrato de QoS com as seguintes informações:

Figura 7.2: Eficiência sem Reconfiguração.



Fonte: Elaborada pelo autor.

QoS_ Contract = [
QoS_ Values = {time: 1.132,0, efficiency: 1,75, cost: 2.264,0}
QoS_ Weigths = {time: 0,8, efficiency: 0,1, cost: 0,1}
Monitor_ Interval = 10,
Reconfiguration_ Interval = 60,
alfa = 0,1]

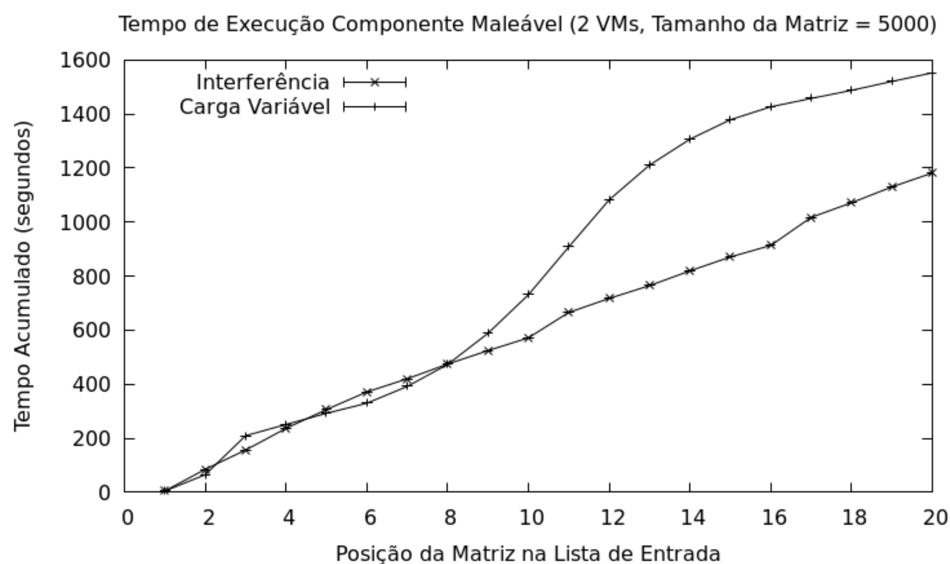
O tempo de execução esperado é de 1132 segundos, que é o valor obtido na Tabela 7.1 para a execução normal com 2 VMs e 20 matrizes de ordem 5000. Os valores obtidos com a mesma configuração inicial são configurados para a eficiência e o custo. Em relação aos pesos, o tempo de execução tem prioridade máxima, enquanto a eficiência e o custo tem a mesma prioridade. O intervalo de monitoramento é configurado para 10 segundos, enquanto no mínimo 60 segundos devem transcorrer entre duas reconfigurações. Por último, a tolerância é definida por um valor de α igual a $0,1$. Submetemos o contrato para execução em um componente de sistema sob o efeito da carga variável da interferência. O comportamento esperado é que o laço de reconfiguração interno ao componente plataforma atue para minimizar os efeitos dos cenários adversos.

Além do contrato, precisamos definir o comportamento das funções gancho do arcabouço proposto: `previsao`, `reconfigurar_parametro` e `atualizarAlfa`. Para a `previsao`, adotamos a *extrapolação linear* como técnica. Por exemplo, se em determinado momento

a variável sensor de progresso na porta de monitoramento do componente computação indicar $0,5$ tendo transcorrido 500 segundos de execução do sistema, a função retorna o valor 1000 segundos como previsão do tempo final de execução. O mesmo cálculo é feito para a previsão do custo, enquanto a eficiência é calculada apenas como o valor da carga no momento da invocação da função pelo laço de controle.

A função `reconfigurar_parametro` define o que fazer para ajustar o valor de cada parâmetro no momento da quebra de contrato. Portanto, de acordo com o contrato adotado, devemos definir as ações de elasticidade para ajustar o tempo de execução, a eficiência e o custo. No caso do tempo de execução, se a previsão estiver abaixo do valor do contrato (execução com melhor desempenho do que o esperado), a ação de elasticidade definida consiste em remover uma máquina virtual do conjunto de recursos. Caso contrário, o tempo acima do valor do contrato implica na adição de uma máquina virtual. Adotamos a mesma estratégia para a eficiência. Entretanto, para o custo, o comportamento é o inverso. Um custo abaixo do esperado nos permite aplicar uma ação de elasticidade horizontal que incrementa o conjunto de recursos. Já se o custo estiver quebrando o limite imposto pelo contrato, devemos remover uma máquina virtual. Devemos reafirmar que as ações são limitadas pelo intervalo ***node_ range*** do contrato e que o comportamento dessa função foi definido a partir do estudo do componente de sistema apresentado na Tabela 7.1. Por último, decidimos não definir um comportamento distinto para a função `atualizarAlfa`. Em outras palavras, o valor do termo α é mantido fixo durante a execução.

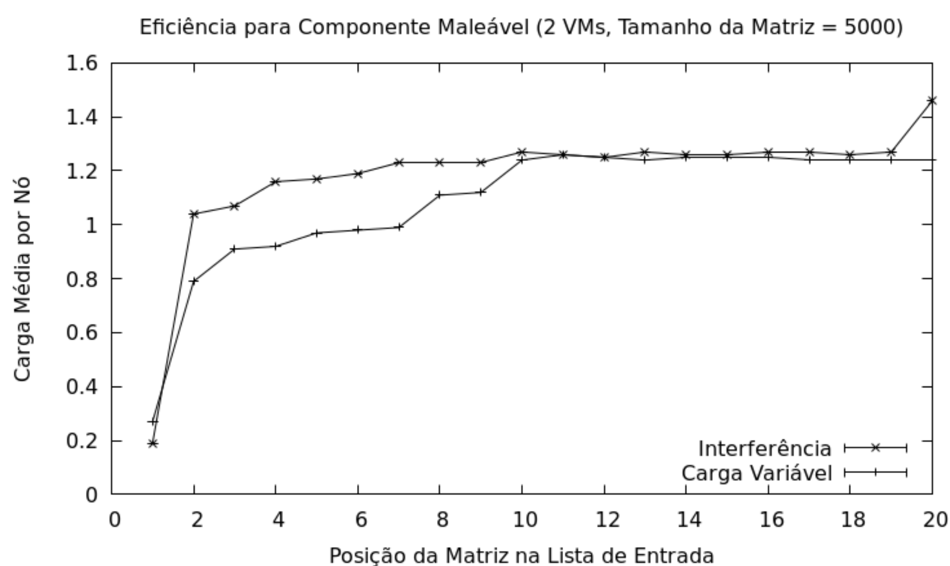
Figura 7.3: Tempo de Execução de Componente de Sistema Maleável com Prioridade de Tempo.



Fonte: Elaborada pelo autor.

Na Figura 7.3 temos o tempo de execução para o componente de sistema limitado pelos cenários e com a reconfiguração maleável ativada. Apesar de não conseguir apresentar o valor exato do contrato, tanto o cenário com interferência quanto com carga variável apresentam tempo total abaixo de 1600 segundos. Para o cenário de interferência, o tempo de execução sem a reconfiguração é de 1675 segundos (ver Figura 7.1). Com a reconfiguração ativada priorizando o tempo, o valor obtido é de 1288 segundos, mais próximo do intervalo descrito no contrato (1132 segundos). Da mesma forma, no cenário da carga variável, a reconfiguração reduz o tempo de execução de 2670 segundos para 1610 segundos. Em ambos os casos, podemos afirmar que o laço de reconfiguração direcionou a execução de acordo com os pesos informados no contrato pelo provedor de aplicações.

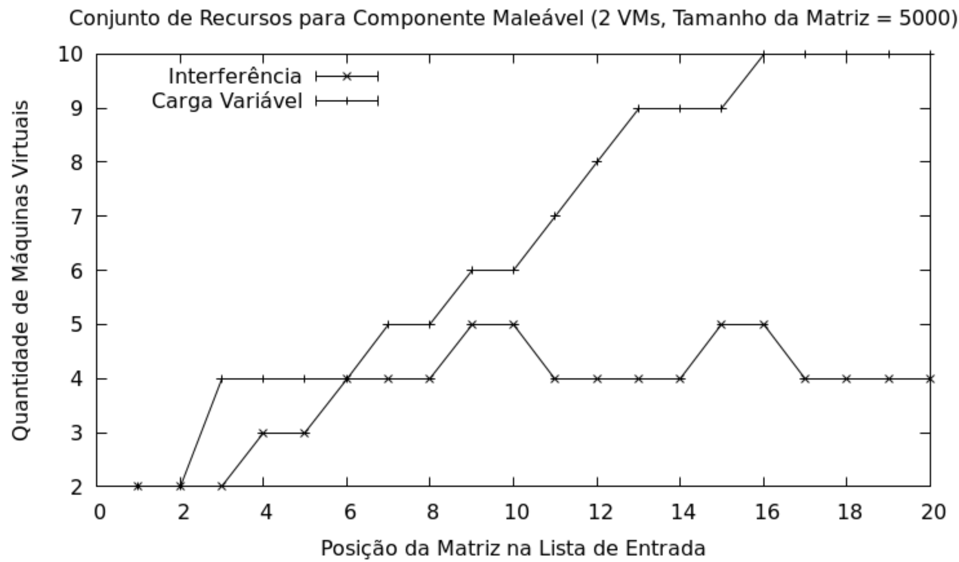
Figura 7.4: Eficiência de Componente de Sistema Maleável com Prioridade de Tempo.



Fonte: Elaborada pelo autor.

Já na Figura 7.4, observamos como a eficiência se comporta no componente de sistema maleável. Como não é prioridade, a eficiência acaba por apresentar valores que desviam do contrato. Em nenhum momento a eficiência chega próximo do valor esperado 1.75, sempre se mantêm abaixo de 1.6 . Como o custo é resultado direto da quantidade de máquinas alocadas, na Figura 7.5 apresentamos a quantidade de recursos alocados em cada entrada da lista de matrizes. Observamos que no caso da carga variável a quantidade de VMs aumenta até o valor limite do contrato, justamente no esforço do ambiente de reconfiguração em reduzir o máximo o tempo de execução. Já o cenário da interferência, que desvia menos do valor do contrato, a reconfiguração estabiliza entre 4 e 5 máquinas virtuais. O cenário de interferência apresenta custo de 5763 unidades monetárias enquanto a carga variável leva o custo a 15346 unidades monetárias. Ambos os valores estão acima do custo de 2264 unidades definido por contrato, mas já que os pesos priorizam o tempo de execução, o ambiente está priorizando o contrato.

Figura 7.5: Custo de Componente de Sistema Maleável com Prioridade de Tempo.



Fonte: Elaborada pelo autor.

O experimento inicial demonstra que a reconfiguração maleável é capaz de priorizar o tempo na execução do componente de sistema. Para demonstrar a mesma habilidade em outros parâmetros, escolhemos o custo para análise e executamos cenários alternando os pesos para o tempo e o custo. Como o cenário da carga variável apresenta maior impacto no comportamento do sistema, decidimos não avaliar a interferência. O resultado pode ser visto na Tabela 7.2. Exceto pelos pesos, em todas as execuções os argumentos do contrato foram os mesmos já definidos.

No início da Tabela 7.2, temos a situação já apresentada, na qual o tempo de execução tem maior prioridade. Percorrendo a tabela, o peso do custo aumenta e a prioridade do tempo de execução diminui. Observamos a tendência de crescimento do tempo enquanto os valores para o custo da execução diminuem, conforme a relação entre os pesos determina. Entretanto, no momento que a prioridade do custo se torna maior do que a do tempo de execução, os valores obtidos para os parâmetros apresentam variação menor. Sendo que optamos por iniciar o componente de sistema com o valor mínimo permitido pelo contrato para o tamanho do conjunto de recursos, a maior prioridade para o custo impede a alocação de novas máquinas virtuais e como esperado o sistema caminha para valores próximos aos apresentados sem a reconfiguração habilitada.

Tabela 7.2: Impacto da Variação dos Pesos no Cenário da Carga Variável.

Pesos	Tempo de Execução	Custo
Custo = 0.1, Tempo = 0.8	1610	15346
Custo = 0.2, Tempo = 0.7	2054	8076
Custo = 0.3, Tempo = 0.6	2195	6965
Custo = 0.4, Tempo = 0.5	2653	6476
Custo = 0.5, Tempo = 0.4	2769	5538
Custo = 0.6, Tempo = 0.3	2718	5437
Custo = 0.7, Tempo = 0.2	2707	5795
Custo = 0.8, Tempo = 0.1	2712	5424
Sem reconfiguração	2575	5150

Fonte: Elaborada pelo autor.

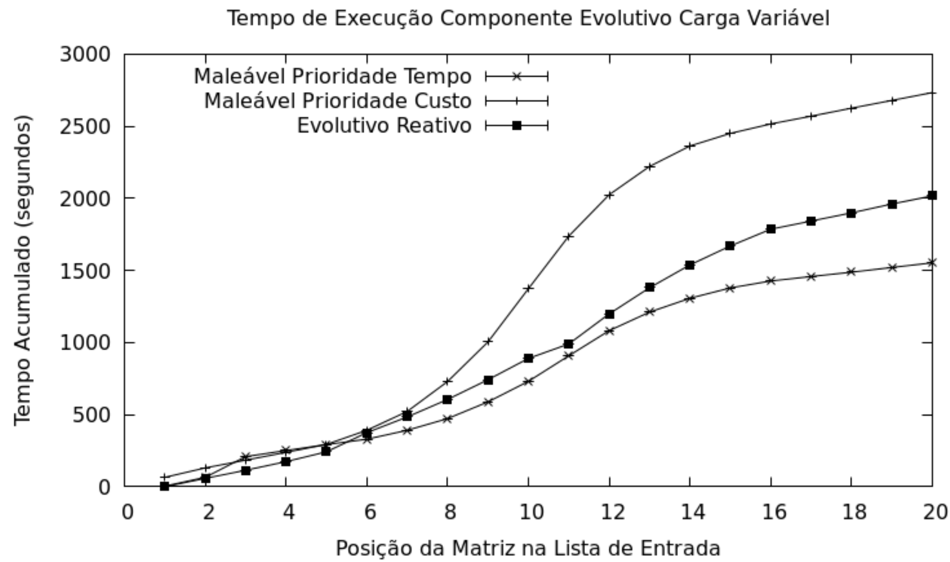
7.3 Sistemas Computacionais Evolutivos

Para análise de computações da espécie reconfigurável evolutiva, desenvolvemos um componente para multiplicação de uma lista de matrizes que analisa o tamanho de cada posição na entrada. Se o tamanho for maior do que a multiplicação anterior, o componente invoca o método `add_node` da API de elasticidade, que por sua vez aciona o componente plataforma através da porta de reconfiguração para adicionar uma máquina virtual ao conjunto de recursos. Caso contrário, uma multiplicação de matrizes com menor ordem acarreta da remoção de uma máquina virtual. O componente de sistema resultante não recupera informações do estado da plataforma para auxiliar na tomada de decisões.

Na Figura 7.6, apresentamos o tempo de execução para os casos maleáveis com maior prioridade de tempo e maior prioridade de custo, ambos já discutidos, o cenário da carga variável. Adicionamos o comportamento do componente de sistema evolutivo com a estratégia discutida. Podemos observar que a espécie evolutiva apresenta melhor tempo de execução (2017 segundos) do que o sistema maleável com prioridade de custo (2733 segundos), porém é mais lento do que o cenário com prioridade de tempo (1533 segundos).

Os dados da Figura 7.7 explicam o desempenho. A execução maleável com prioridade de tempo percebe os efeitos da entrada mal definida e aloca máquinas, sendo que o conjunto expandido de recursos é mantido até o fim da execução com o objetivo de apresentar valores mais próximos possíveis do contrato. Quando o custo é priorizado, nenhuma alocação ocorre, como é esperado. O sistema evolutivo aloca máquinas de acordo com sua lógica interna de avaliação da entrada. Como não é guiado por contrato, assim que a carga se normaliza, os recursos adicionais são liberados. Dessa forma, as últimas entradas na lista, multiplicações de matrizes de ordem 5000, são executados por apenas 2

Figura 7.6: Tempo de Execução do Componente Evolutivo.



Fonte: Elaborada pelo autor.

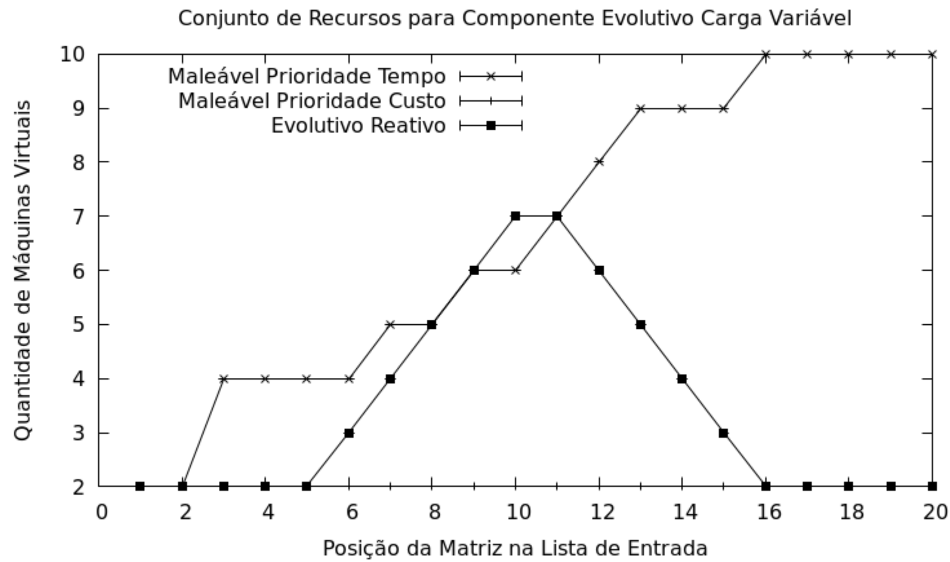
VMs, enquanto no caso maleável com prioridade de tempo os mesmos problemas utilizam todos os recursos possíveis pelo contrato da plataforma.

7.4 Conclusões

Neste capítulo, apresentamos estudos de casos para sistemas formados por componentes de computação da nova espécie *reconfiguráveis* proposta como extensão ao conjunto de espécies da HPC Shelf. Para o caso maleável, partimos de uma análise empírica do componente implementado que serviu como base para definir o comportamento das funções do *framework* proposto. Feita a comparação com a execução sem reconfiguração, demonstramos que o *framework* adapta a execução em busca do cumprimento do contrato de QoS, obedecendo as prioridades definidas pelo provedor de aplicações através dos pesos. Já para o caso evolutivo, demonstramos que o desenvolvedor de componentes pode implementar uma lógica interna ao componente para guiar a reconfiguração.

A experiência no desenvolvimentos dos componentes nos leva a concluir que a espécie evolutiva não é a melhor opção em termos de reuso de código e separação dos interesses. O caminho adequado para o registro de componentes da espécie reconfigurável envolve compreensão do comportamento do código pelo desenvolvedor, seja pela análise empírica ou de isoeficiência, seguida da definição das funções `previsao` e `reconfigurar_parametro`. O provedor, ao construir sua aplicação no SAFe, sabe o local exato do estabelecimento das ações de elasticidade no código do componente da computação e pode ajustar os pesos

Figura 7.7: Custo de Componente Evolutivo.



Fonte: Elaborada pelo autor.

de maneira a otimizar a execução. Entretanto, é importante que a HPC Shelf suporte a subespécie evolutiva, visto que a Computação de Alto Desempenho possui quantidade considerável de código legado, construído sem as técnicas de modelagem modernas como orientação a objetivos. Em um momento inicial, o desenvolvedor de componentes pode migrar seu código adicionando chamadas à API de elasticidade. Com o decorrer da utilização de seu componente, haverá o incentivo para remodelá-lo com o objetivo de tirar o maior proveito das habilidades da nuvem de serviços. Por último, não podemos eliminar a possibilidade de problemas nos quais o controle com fina granularidade da elasticidade não pode ser separado da lógica do código. Em ambos os casos, com as extensões propostas nesta Tese, a HPC Shelf apresentar maior valor agregado para definição de um ambiente para execução de componentes de aplicações de alto desempenho.

8. Conclusão

Em um cenário com vários componentes de sistemas computacionais compatíveis, a pesquisa em torno da HPC Shelf busca desenvolver mecanismos que permitam selecionar as opções de acordo com requisitos funcionais e as classifica obedecendo requisitos não funcionais de qualidade de serviço. Esses requisitos são fornecidos pela aplicação desenvolvida com o *framework* SAFe, representando os interesses do provedor de aplicação e usuário final. Mesmo após a execução dos algoritmos no Core, o contrato de QoS não está garantido devido a imprecisões na definição dos requisitos por parte do provedor e usuário, além da possibilidade de alteração do estado dos recursos do componente plataforma.

Nesta Tese, propusemos modelar um arcabouço para definição de mecanismos de reconfiguração elástica capazes de adaptar a execução de componentes de sistemas paralelos em direção ao cumprimento dos requisitos presentes no contrato de QoS. Atendemos os objetivos específicos ao levantar o estado da arte em elasticidade e programação baseada em componentes para CAD e usamos tal estudo como subsídio para estender o mecanismo existente de classificação da HPC Shelf com uma técnica específica para suporte a definição de prioridades entre os parâmetros de QoS. Desse esforço partimos para propor uma arquitetura de componentes de sistema da espécie reconfigurável que habilita os atores da HPC Shelf a colaborarem no controle do ciclo de vida de uma aplicação de Computação de Alto Desempenho executando em infraestruturas computacionais modernas (*clusters* e nuvens).

Os estudos de casos demonstram que o arcabouço modelado é sensível aos pesos que definem as prioridades entre os parâmetros suportados. Portanto, sustentamos que o ambiente resultante atende à hipótese levantada no objetivo principal e se caracteriza como uma solução PaaS, já que os atores da nuvem de serviços não precisam lidar manualmente com a alocação e liberação de recursos, pois os componentes de sistemas reconfiguráveis encapsulam a lógica responsável pelas ações de reconfiguração elástica. O resultado final é um mecanismo potencialmente útil para uma nuvem de serviços de Computação de Alto Desempenho com suporte a elasticidade de computação paralela. Listamos as principais contribuições da Tese na Seção 8.1.

Apesar da modelagem da arquitetura apresentar viabilidade, ainda persistem desafios que servem como fomento para trabalhos futuros. Discutimos esses problemas na Seção 8.2.

8.1 Contribuições

As principais contribuições da Tese são:

- Levantamento do estado da arte de elasticidade para computações de alto desempenho (Capítulo 4);
- A partir do arcabouço de seleção e classificação de componentes baseados em contratos contextuais da HPC Shelf, chamado Alite (objeto de estudo e contribuição de uma outra tese de doutorado do mesmo grupo de pesquisa), definimos uma técnica particular para classificação dos sistemas e definição do contrato de QoS (Seção 5.7), com vistas a validar as contribuições desta Tese;
- Um arcabouço que adiciona a espécie de computações *reconfiguráveis*, suportando os aspectos maleáveis e evolutivos (Capítulo 6);
- Estudos de caso que comprovam a viabilidade do arcabouço proposto (Capítulo 7).

8.2 Desafios

A seguir, resumimos as principais limitações da Tese que servem de base para os trabalhos futuros.

8.2.1 Reputação de Componentes

Com as informações do contrato, a função de estimativa e os mecanismos de reconfiguração, podemos garantir que a HPC Shelf direciona a execução de componentes para o cumprimento dos requisitos de QoS do usuário especialista. Entretanto, mesmo com as correções aplicadas em função de erros de estimativa e imprevisibilidade da infraestrutura, a execução pode falhar em cumprir o contrato. Isso afeta a reputação tanto do desenvolvedor de componentes quando do mantenedor da plataforma. Uma maneira de atenuar o problema seria atribuir um valor de *reputação* para os atores. Quebras de contratos de sistemas computacionais teriam valor negativo na reputação do ator, enquanto o cumprimento do contrato, mesmo com o uso de reconfigurações, teria efeito positivo. Dessa forma, além de permitir que o provedor escolha sistemas formados por componentes de mantenedores e desenvolvedores com melhor reputação, o próprio ator pode trabalhar para melhorar sua avaliação através da otimização dos componentes que fornece à HPC Shelf. Por exemplo, o desenvolvedor de componentes pode aprimorar um algoritmo para

apresentar maior precisão e previsibilidade no uso dos recursos, enquanto um mantenedor busca implementar um sistema de reserva de recursos mais eficaz.

Outra questão relacionada a reputação deriva do fato que o mantenedor de plataforma tem requisitos próprios que podem entrar em conflito com a qualidade de serviço exigida pelo provedor de aplicações. Por exemplo, um mantenedor pode desejar manter a vazão de sistemas computacionais executados por unidade de tempo dentro de uma faixa de valores. Para tal, em determinado momento ele pode priorizar a execução de sistemas de curta duração em detrimento de componentes que executam por intervalos maiores. Buscar o equilíbrio entre os interesses conflitantes dos atores é um problema da *teoria do jogos*, sendo já existem trabalhos que abordam o problema através de modelagem analítica para infraestruturas computacionais (TENG; MAGOULÈS, 2010) (WEI, 2010).

Para suportar um sistema de reputação na HPC Shelf, precisamos incluir no serviço Core a capacidade de armazenar um histórico de execução para cada componente de sistema. Esse histórico deve conter o contrato submetido e o valor dos parâmetros que de fato foram apresentados durante a execução. De posse dessa informação, uma *função de reputação* pode calcular uma valor de reputação para cada ator envolvido no estabelecimento do sistema. O mesmo histórico pode servir como base para execução de uma *função de arbitração* no Core, que busca definir e classificar componentes de sistema de acordo com o equilíbrio dos interesses conflitantes dos atores da HPC Shelf. Essas duas funções não são o foco da presente Tese, entretanto o *framework* apresentado é um alicerce necessário para essas futuras extensões na HPC Shelf.

8.2.2 Avaliação de Padrões de Paralelismo e Plataformas

Os estudos de caso apresentados no Capítulo 7 comprovam a viabilidade do arcabouço. Entretanto, as comparações feitas são realizadas entre um ambiente sem reconfiguração e um protótipo construído a partir da solução proposta nesta Tese. As diversas técnicas discutidas no Capítulo 4 não são implementadas e avaliadas no *framework*. Isso não foi feito porque as soluções existentes não adotam uma arquitetura de componentes, sendo que estão fortemente acopladas às infraestruturas de nuvens e *clusters* utilizadas na sua implantação. Como trabalho futuro, as técnicas existentes serão implementadas no *framework* através de componentes abstratos. Além de expandir a funcionalidade da nuvem de serviços, esse trabalho permitirá maior validação do *framework*.

Outro aspecto que merece maior estudo é o padrão de paralelismo avaliado. Usamos uma aplicação da álgebra linear computacional que representa uma classe de paralelismo importante, sendo adotada até como *benchmark* para infraestruturas. Entretanto, computações com padrões distintos de paralelismo, incluindo aquelas nas quais o gargalo é

a comunicação ou acesso à disco, também são importantes para a Computação de Alto Desempenho e podem fornecer uma análise mais criteriosa ao serem implementas no *framework*. Por exemplo, não utilizamos o parâmetro de acurácia nos estudos de casos e não nos aprofundamos na análise da eficiência, dois cenários que computações diferentes da que utilizamos podem exigir. Portanto, definimos como trabalhos futuros implementar uma gama mais diversa de aplicações na HPC Shelf.

Referências Bibliográficas

ALDINUCCI, M.; CAMPA, S.; DANELUTTO, M.; VANNESCHI, M. Behavioural Skeletons in GCM: Autonomic Management of Grid Components. In: IEEE. *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. [S.l.], 2008. p. 54–63.

AMAZON. Amazon Elastic Compute Cloud (Amazon EC2). 2010. Disponível em: <http://aws.amazon.com>.

ANDRADE, N.; CIRNE, W.; BRASILEIRO, F.; ROISENBERG, P. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In: *Job scheduling strategies for parallel processing*. [S.l.]: Springer, 2003. p. 61–86.

ARMSTRONG, R.; KUMFERT, G.; MCINNES, L. C.; PARKER, S.; ALLAN, B.; SOTTILE, M.; EPPERLY, T.; DAHLGREN, T. The CCA Component Model for High-Performance Scientific Computing. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 18, n. 2, p. 215–229, 2006.

BAUDE, F.; CAROMEL, D.; DALMASSO, C.; DANELUTTO, M.; GETOV, V.; HENRIO, L.; PÉREZ, C. GCM: A Grid Extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications-annales des télécommunications*, Springer, v. 64, n. 1-2, p. 5–24, 2009.

BENZ, K.; BOHNERT, T. M. Elastic Scaling of Cloud Application Performance Based on Western Electric Rules by Injection of Aspect-oriented Code. *Procedia Computer Science*, v. 61, p. 198 – 205, 2015. ISSN 1877-0509. Complex Adaptive Systems San Jose, CA November 2-4, 2015. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915030239>.

BRUNETON, E.; COUPAYE, T.; LECLERCQ, M.; QUÉMA, V.; STEFANI, J.-B. The Fractal Component Model and its Support in Java. *Software: Practice and Experience*, Wiley Online Library, v. 36, n. 11-12, p. 1257–1284, 2006.

CABALLER, M.; ALFONSO, C. de; ALVARRUIZ, F.; MOLTó, G. EC3: Elastic Cloud Computing Cluster. *Journal of Computer and System Sciences*, v. 79, n. 8, p. 1341–1351, dez. 2013. ISSN 00220000. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0022000013001141>.

CABALLER, M.; ALFONSO, C. de; MOLTó, G.; ROMERO, E.; BLANQUER, I.; GARCíA, A. CodeCloud: A Platform to Enable Execution of Programming Models on the Clouds. *Journal of Systems and Software*, v. 93, n. 0, p. 187 – 198, 2014. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121214000521>.

CARVALHO JUNIOR, F. H. de; CORREA, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In: *Workshop on Component-Based High Performance Computing (CBHPC'2010)*. [S.l.]: IEEE, 2010. p. 339–348. ISBN 9781424493470.

CARVALHO JUNIOR, F. H. de; LINS, R. D. An Institutional Theory for#-Components. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 195, p. 113–132, 2008.

CARVALHO JUNIOR, F. H. de; LINS, R. D.; CORRÊA, R. C.; ARAÚJO, G. A. Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice & Experience*, v. 19, n. 5, p. 697–719, 2007.

CARVALHO JUNIOR, F. H. de; REZENDE, C. A. A Case Study on Expressiveness and Performance of Component-Oriented Parallel Programming. *Journal of Parallel and Distributed Computing*, v. 73, n. 5, p. 557–569, 2013. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731512002882>.

CARVALHO JUNIOR, F. H. de; REZENDE, C. A.; SILVA, J. C.; ALAM, W. G. Al; ALENCAR, J. M. U. de. Contextual Abstraction in a Type System for Component-based High Performance Computing Platforms. *Science of Computer Programming*, v. 132, p. 96–128, 2016. ISSN 0167-6423.

CATLETT, C.; ALLCOCK, W. E.; ANDREWS, P.; AYDT, R. A.; BAIR, R.; BALAC, N.; BANISTER, B.; BARKER, T.; BARTELT, M.; BECKMAN, P. H. et al. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In: *High Performance Computing Workshop*. [S.l.: s.n.], 2006. p. 225–249.

CHEN, H.-b.; TORREZ, A.; PARKS FIELDS, J. C. F.; ILLESCAS, D.; PEREZ-MEDINA, R.; LAFON, J.; HAYNES, B.; HERRERA, J. *Performance Analysis and Evaluation of LANL's PaScalBB I/O nodes using Quad-Data-Rate Infiniband and Multiple 10-Gigabit Ethernets Bonding*. [S.l.], 2011. Disponível em: <http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-11-03023>.

COUTINHO, E. F.; CARVALHO SOUSA, F. R. de; REGO, P. A. L.; GOMES, D. G.; SOUZA, J. N. de. Elasticity in Cloud Computing: A Survey. *Annals of Telecommunications-Annales des Télécommunications*, Springer Paris, p. 1–21, 2015.

COUTINHO, E. F.; GOMES, D. G.; SOUZA, J. N. de. Elasticity Provisioning in Hybrid Computational Clouds Based on Autonomic Computing Concepts. *V Workshop de Sistemas Distribuídos Autonômicos (WOSIDA2015), At Vitória / ES / Brazil*, 2015.

DAGUM, L.; MENON, R. OpenMP: An Industry Standard API for Shared-memory Programming. *Computational Science & Engineering, IEEE*, IEEE, v. 5, n. 1, p. 46–55, 1998.

DONGARRA, J. *Visit to the National University for Defense Technology Changsha China*. 2013. Disponível em: <http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf>.

DURAN, A.; KLEMM, M. The Intel® Many Integrated Core Architecture. In: *2012 International Conference on High Performance Computing and Simulation (HPCS)*. [S.l.]: IEEE Computer Society, 2012. p. 365–366. ISBN 978-1-4673-2359-8.

EL MAGHRAOUI, K.; DESELL, T. J.; SZYMANSKI, B. K.; VARELA, C. A. Malleable Iterative MPI Applications. *Concurrency and Computation: Practice and Experience*, John Wiley And Sons,Ltd., v. 21, n. 3, p. 393–413, 2009. ISSN 1532-0634. Disponível em: <http://dx.doi.org/10.1002/cpe.1362>.

FAN, P.; CHEN, Z.; WANG, J.; ZHENG, Z.; LYU, M. R. Topology-Aware Deployment of Scientific Applications in Cloud Computing. In: IEEE. *Cloud Computing (CLOUD), 2012 IEEE 5th international conference on*. [S.l.], 2012. p. 319–326.

FAN, Z.; QIU, F.; KAUFMAN, A.; STOVER, S. Yoakum. GPU Cluster for High Performance Computing. In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)*. [S.l.]: IEEE Computer Society, 2004. p. 47–47. ISBN 0-7695-2153-3.

FEITELSON, D.; RUDOLPH, L. Toward Convergence in Job Schedulers for Parallel Supercomputers. Springer Berlin Heidelberg, v. 1162, p. 1–26, 1996. Disponível em: <http://dx.doi.org/10.1007/BFb0022284>.

FOSTER, I.; KESSELMAN, C. *The Grid 2: Blueprint for a new computing infrastructure*. [S.l.]: Elsevier, 2003.

GALANTE, G.; BONA, L. Supporting Elasticity in OpenMP Applications. In: *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. [S.l.: s.n.], 2014. p. 188–195. ISSN 1066-6192.

GALANTE, G.; BONA, L. C. E. D. A Programming-Level Approach for Elasticizing Parallel Scientific Applications. *Journal of Systems and*

Software, v. 110, p. 239 – 252, 2015. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121215001946>.

GALANTE, G.; BONA, L. de. A Survey on Cloud Computing Elasticity. In: *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*. [S.l.: s.n.], 2012. p. 263–270.

GARCIA, C. E.; PRETT, D. M.; MORARI, M. Model Predictive Control: Theory and Practice — A Survey. *Automatica*, Elsevier, v. 25, n. 3, p. 335–348, 1989.

GELADO, I.; STONE, J. E.; CABEZAS, J.; PATEL, S.; NAVARRO, N.; HWU, W.-m. W. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2010. (ASPLOS XV), p. 347–358. ISBN 978-1-60558-839-1. Disponível em: <http://doi.acm.org/10.1145/1736020.1736059>.

GIL, Y. From Data to Knowledge to Discoveries: Scientific Workflows and Artificial Intelligence. *Scientific Programming*, v. 16, n. 4, 2008.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel computing*, Elsevier, v. 22, n. 6, p. 789–828, 1996.

GUPTA, A. Techniques for Efficient High Performance Computing in the Cloud. 2014. Disponível em: <http://hdl.handle.net/2142/50718>.

HALL, M. W.; GIL, Y.; LUCAS, R. F. Self-Configuring Applications for Heterogeneous Systems: Program Composition and Optimization using Cognitive Techniques. *Proceedings of the IEEE*, IEEE, v. 96, n. 5, p. 849–862, 2008.

HAMILTON, J. D. *Time Series Analysis*. [S.l.]: Princeton University Press, 1994.

HERBORDT, M. C.; VANCOURT, T.; GU, Y.; SUKHWANI, B.; CONTI, A.; MODEL, J.; DISABELLO, D. Achieving High Performance with FPGA-Based Computing. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 40, p. 50–57, March 2007. ISSN 0018-9162. Disponível em: <http://dl.acm.org/citation.cfm?id=1251558.1251716>.

HUANG, H.; WANG, L.; TAK, B. C.; WANG, L.; TANG, C. CAP3: A Cloud Auto-Provisioning Framework for Parallel Processing Using On-Demand and Spot Instances. In: *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013. p. 228–235. ISBN 978-0-7695-5028-2. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6676699>.

IPEK, E.; SUPINSKI, B. R. de; SCHULZ, M.; MCKEE, S. A. An Approach to Performance Prediction for Parallel Applications. In: *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2005. (Euro-Par'05), p. 196–205. ISBN 3-540-28700-0, 978-3-540-28700-1. Disponível em: <http://dx.doi.org/10.1007/11549468_24>.

JORISSEN, K.; VILA, F. D.; REHR, J. J. A High Performance Scientific Cloud Computing Environment for Materials Simulations. *Computer Physics Communications*, Elsevier, v. 183, n. 9, p. 1911–1919, 2012.

KEPHART, J.; KEPHART, J.; CHESS, D.; BOUTILIER, C.; DAS, R.; KEPHART, J. O.; WALSH, W. E. An Architectural Blueprint for Autonomic Computing. *IEEE internet computing*, v. 18, n. 21, 2007.

KLEINROCK, L. *Queueing systems, volume 2: Computer applications*. [S.l.]: Wiley New York, 1976.

KUMAR, V.; GRAMA, A.; GUPTA, A.; KARYPIS, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. [S.l.]: Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

LEDYAYEV, R.; RICHTER, H. High Performance Computing in a Cloud Using OpenStack. In: *CLOUD COMPUTING 2014, The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization*. [S.l.: s.n.], 2014. p. 108–113.

LORIDO-BOTRAN, T.; MIGUEL-ALONSO, J.; LOZANO, J. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, Springer Netherlands, v. 12, n. 4, p. 559–592, 2014. ISSN 1570-7873. Disponível em: <http://dx.doi.org/10.1007/s10723-014-9314-7>.

MARIN, G.; MELLOR-CRUMMEY, J. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2004. (SIGMETRICS '04/Performance '04), p. 2–13. ISBN 1-58113-873-3. Disponível em: <http://doi.acm.org/10.1145/1005686.1005691>.

MATEESCU, G.; GENTZSCH, W.; RIBBENS, C. J. Hybrid Computing—Where HPC meets grid and Cloud Computing. *Future Generation Computer Systems*, v. 27, n. 5, p. 440–453, maio 2011. ISSN 0167739X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X1000213X>.

MAURER, M.; BRESKOVIC, I.; EMEAKAROHA, V.; BRANDIC, I. Revealing the MAPE Loop for the Autonomic Management of Cloud Infrastructures. In: *Computers*

and Communications (ISCC), 2011 IEEE Symposium on. [S.l.: s.n.], 2011. p. 147–152. ISSN 1530-1346.

MCINNES, L. C.; ALLAN, B. A.; ARMSTRONG, R.; BENSON, S. J.; BERNHOLDT, D. E.; DAHLGREN, T. L.; DIACHIN, L. F.; KRISHNAN, M.; KOHL, J. A.; LARSON, J. W. et al. Parallel PDE-based Simulations Using the Common Component Architecture. In: *Numerical Solution of Partial Differential Equations on Parallel Computers*. [S.l.]: Springer, 2006. p. 327–381.

MCINNES, L. C.; RAY, J.; ARMSTRONG, R.; DAHLGREN, T. L.; MALONY, A.; NORRIS, B.; SHENDE, S.; KENNY, J. P.; STEENSLAND, J. *Computational Quality of Service for Scientific CCA applications: Composition, Substitution, and Reconfiguration*. [S.l.], 2006. Disponível em: <ftp://140.221.6.23/pub/techreports/reports/P1326.pdf>.

MELL, P.; GRANCE, T. The NIST Definition of Cloud Computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011.

MENCAGLI, G.; VANNESCHI, M.; VESPA, E. Control-Theoretic Adaptation Strategies for Autonomic Reconfigurable Parallel Applications on Cloud Environments. In: *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. [S.l.: s.n.], 2013. p. 11–18.

MUELLER, F. et al. A Library Implementation of POSIX Threads under UNIX. In: *USENIX Winter*. [S.l.: s.n.], 1993. p. 29–42.

NIU, S.; ZHAI, J.; MA, X.; TANG, X.; CHEN, W. Cost-effective Cloud HPC Resource Provisioning by Building Semi-Elastic Virtual Clusters. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. New York, New York, USA: ACM Press, 2013. p. 1–12. ISBN 9781450323789. Disponível em: <http://dl.acm.org/citation.cfm?id=2503210.2503236>.

R. RIGHI, R. d.; RODRIGUES, V. F.; COSTA, C. A. da; GALANTE, G.; BONA, L. C. E. de; FERRETO, T. AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud. *IEEE Transactions on Cloud Computing*, v. 4, n. 1, p. 6–19, Jan 2016. ISSN 2168-7161.

RAVEENDRAN, A.; BICER, T.; AGRAWAL, G. A Framework for Elastic Execution of Existing MPI Programs. p. 940–947, May 2011. ISSN 1530-2075.

ROSA RIGHI, R. da; RODRIGUES, V. F.; ROSTIROLLA, G.; COSTA, C. A. da; ROLOFF, E.; NAVAUX, P. O. A. A Lightweight Plug-and-Play Elasticity Service for Self-Organizing Resource Provisioning on Parallel Applications. *Future Generation Computer Systems*, Elsevier, 2017.

ROY, N.; DUBEY, A.; GOKHALE, A. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In: *2011 IEEE 4th International Conference on Cloud Computing*. [S.l.: s.n.], 2011. p. 500–507. ISSN 2159-6182.

RUIVO, T. P. P. D. L.; ALTAYO, G. B.; GARZOGLIO, G.; TIMM, S.; KIM, H. W.; NOH, S.-Y.; RAICU, I. Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. [S.l.], 2014. p. 943–948.

RUSSELL, R. M. The CRAY-1 Computer System. *Communications of the ACM*, ACM, v. 21, n. 1, p. 63–72, 1978.

SEGAL, B.; ROBERTSON, L.; GAGLIARDI, F.; CARMINATI, F.; CERN, G. Grid Computing: The European Data Grid Project. In: *IEEE Nuclear Science Symposium and Medical Imaging Conference*. [S.l.: s.n.], 2000. v. 1, p. 2.

SHENDE, S. S.; MALONY, A. D. The TAU Parallel Performance System. *The International Journal of High Performance Computing Applications*, Sage Publications Sage CA: Thousand Oaks, CA, v. 20, n. 2, p. 287–311, 2006.

SHUDLER, S.; CALOTOIU, A.; HOEFLER, T.; WOLF, F. Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications. In: ACM. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. [S.l.], 2017. p. 131–143.

SILVA, J. C.; CARVALHO JUNIOR, F. H. de. A Platform of Scientific Workflows for Orchestration of Parallel Components in a Cloud of High Performance Computing Applications. In: *Lecture Notes in Computer Science*: Proceedings of the XX Brazilian Symposium on Programming Languages (SBLP'2016). [S.l.]: Springer, 2016. v. 9889, p. 156–170.

SOUSA, F. R. C.; MACHADO, J. C. Towards Elastic Multi-Tenant Database Replication with Quality of Service. In: *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. Washington, DC, USA: IEEE Computer Society, 2012. (UCC '12), p. 168–175. ISBN 978-0-7695-4862-3. Disponível em: <http://dx.doi.org/10.1109/UCC.2012.36>.

STERLING, T. L. *Beowulf Cluster Computing with Linux*. [S.l.]: MIT press, 2002.

SUTTON, R. S.; BARTO, A. G. *Introduction to Reinforcement Learning*. [S.l.]: MIT Press Cambridge, 1998.

TANENBAUM, A. S. *Modern Operating Systems*. [S.l.]: Prentice hall Englewood Cliffs, 1992.

TENG, F.; MAGOULÈS, F. A New Game Theoretical Resource Allocation Algorithm for Cloud Computing. In: _____. *Advances in Grid and Pervasive Computing: 5th International Conference, GPC 2010, Hualien, Taiwan, May 10-13, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 321–330. ISBN 978-3-642-13067-0.

WALKER, E. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *Usenix Login*, v. 33, n. 5, p. 18–23, 2008.

WANG, A. J. A.; QIAN, K. *Component-oriented programming*. [S.l.]: John Wiley & Sons, 2005.

WEI, G.; VASILAKOS, A. V.; ZHENG, Y.; XIONG, N. A Game-Theoretic Method of Fair Resource Allocation for Cloud Computing Services. *The Journal of Supercomputing*, v. 54, n. 2, p. 252–269, Nov 2010. ISSN 1573-0484. Disponível em: <http://dx.doi.org/10.1007/s11227-009-0318-1>.

YOO, A. B.; JETTE, M. A.; GRONDONA, M. SLURM: Simple Linux Utility for Resource Management. In: SPRINGER. *Job Scheduling Strategies for Parallel Processing*. [S.l.], 2003. p. 44–60.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud Computing: State-of-the-art and Research Challenges. *Journal of internet services and applications*, Springer, v. 1, n. 1, p. 7–18, 2010.