UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

Marcio Rodrigo Melo Martins

Uma Abordagem para o Gerenciamento da Execução de Aplicações com Restrições de Tempo de Execução em Grades Computacionais Oportunistas

São Luís
2012

Marcio Rodrigo Melo Martins

Uma Abordagem para o Gerenciamento da Execução de Aplicações com Restrições de Tempo de Execução em Grades Computacionais Oportunistas

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão como requisito parcial para a obtenção do grau de MESTRE em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva

Doutor em Ciência da Computação – UFMA

São Luís

2012

Melo Martins, Marcio Rodrigo

Uma Abordagem para o Gerenciamento da Execução de Aplicações com Restrições de Tempo de Execução em Grades Computacionais Oportunistas / Marcio Rodrigo Melo Martins. – São Luís, 2012.

100 f.

Orientador: Francisco José da Silva e Silva.

Impresso por computador (fotocópia).

Dissertação (Mestrado) – Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia de Eletricidade. São Luís, 2012.

1. Estratégias de escalonamento. 2. Simulação. 3. Grades Computacionais Oportunistas. 4.Aplicações Soft-deadline. I. da Silva e Silva, Francisco José, orient. II. Título.

CDU 004.75

UMA ABORDAGEM PARA O GERENCIAMENTO DA EXECUÇÃO DE APLICAÇÕES COM RESTRIÇÕES DE TEMPO DE EXECUÇÃO EM GRADES COMPUTACIONAIS OPORTUNISTAS

Marcio Rodrigo Melo Martins

Dissertação aprovada em 13 de abril de 2012.

Prof. Francisco José da Silva e Silva, Dr.
(Orientador)

Prof. Omar Andres Carmona Cortes, Dr.
(Membro da Banca Examinadora)

Prof. Zair Abdelouahab, Ph.D.
(Membro da Banca Examinadora)

Prof. Alexandre César Tavares Vidal, Dr.
(Membro da Banca Examinadora)

Aos meus pais, meus irmãos, meu filho, minha família, meus amigos e meus professores.

Resumo

Um ambiente de computação de grade oportunista aproveita ciclos ociosos de computadores e estações de trabalho que podem ser distribuídos por vários domínios administrativos para a execução de aplicações de alto desempenho. Grades oportunistas geralmente são construídas a partir de computadores pessoais que não precisam ser dedicados para a execução de aplicações em grade. Neste tipo de grade, a carga de trabalho deve coexistir com execuções de aplicações locais submetidos pelos usuários dos nós que a compõe. Assim, seu ambiente de execução é tipicamente dinâmico, heterogêneo e imprevisível e falhas ocorrem com frequência. Além disso, os recursos de uma grade oportunista podem ser usados a qualquer momento para a execução de tarefas locais, o que torna difícil prever a conclusão das tarefas em execução nos nós da grade. Essas características dificultam a execução bem sucedida de aplicações para as quais existem restrições de tempo relacionada com a sua conclusão.

Este trabalho apresenta um mecanismo de gerenciamento da execução de aplicações projetado especificamente para ambientes de computação de grade oportunista cujas aplicações possuem prazos de execução (*deadline*) definidos pelos usuários durante sua submissão ao sistema. O mecanismo proposto é baseado em uma abordagem dinâmica de escalonamento e reescalonamento de aplicações e foi avaliado através de um modelo de simulação levando-se em consideração vários cenários típicos de grades oportunistas. Os resultados demonstraram os benefícios da abordagem proposta em comparação com abordagens de escalonamento de aplicações tradicionalmente utilizadas em grades oportunistas.

Palavras-chave: Estratégias de escalonamento. Simulação. Grades Computacionais Oportunistas. Aplicações Soft-deadline.

Abstract

An opportunistic grid computing environment takes advantage of idle computing cycles of regular computers and workstations that can be spread across several administrative domains for running high performance applications. Opportunistic grids are usually constructed from personal computers that do not need to be dedicated for executing grid applications. The grid workload must coexist with local applications executions, submitted by the nodes regular users. Thus, its execution environment is typically dynamic, heterogeneous and unpredictable failures occur frequently. In addition, the resources of an opportunistic grid can be used at any time for the execution of local tasks, making it difficult to preview the conclusion of the tasks running on the grid nodes. These characteristics hinder the successful execution of applications for which there are time restrictions related to its completion.

This thesis presents a management mechanism specifically designed for opportunistic grid computing environments for handling the execution of applications with time deadlines set by users during their submission to the system. The proposed mechanism is based on a dynamic scheduling and rescheduling approach and was evaluated using a simulated model considering various typical scenarios of opportunistic grids. The results demonstrated the benefits of the proposed approach in comparison to traditional scheduling approaches applied in opportunistic grids.

Keywords: *Scheduling strategies. Simulation. Opportunistic Grids. Applications Soft-deadline.*

Agradecimentos

Agradeço primeiramente a Deus por estar sempre me abençoando e por me dar saúde e motivação, para que eu pudesse realizar este trabalho. Sem ele nada seria possível.
Aos meus pais, pelo exemplo de família que eles são e pela criação direcionada a este caminho.
Ao meu filho David Rodrigo que é minha maior motivação. E a mãe de David pela excelente criação dada a ele na minha ausência como pai.
Agradeço ao meu orientador, Prof. Dr. Francisco José da Silva e Silva, por ter acreditado em mim, apesar das minhas deficiências como orientando, por corrigir meus erros e por mostrar-se sempre preocupado com seus orientandos, nunca nos deixando desistir.
Não posso deixar de agradecer mais uma vez meu orientador pois graças a administração do Laboratório de Sistemas Distribuídos (LSD) que me foi dada por Francisco durante o mestrado, adquiri conhecimentos práticos e fundamentais durante a administração do Laboratório que se tornaram essenciais para minha aprovação em quarto lugar no concurso para Técnico de Tecnologia da Informação na UFMA - Campus São Luís.
Agradeço especialmente a equipe de grades autonômicas composta por Berto, Eduardo, Jesseildo e Igor Ramos, pois cada um se dedicou e contribuiu para que este trabalho fosse concluído.
Agradeço infinitamente a Berto de Tácio Pereira Gomes e Jesseildo Figueredo Gonçalves que acompanharam de perto meu trabalho e sem ajuda deles não teria conseguido, agradeço também o suporte no AGST que Berto proporcionou para que este trabalho tivesse exito.
Não posso deixar de agradecer também aos pesquisadores Alcides Costa, Alcilene, Ariel, Diego, Dejailson, Domingos, Felipe Viveiros, Igor Leonardo, Jaciara, Jackson Amaral, Jansen, Lucena, Luis Carlos, Marcondes, Marlon, Rafael Araujo, Regilaine, Romulo, Samir Fahd, Thiago Mendes e os professores Luciano e Samyr todos os integrantes do Laboratório de Sistemas Distribuídos (LSD - UFMA), pela companhia

nos momentos de aflição e de descontração e com os quais compartilhamos conhecimentos.
Agradeço a Diego por ter me ajudado a resolver vários problemas de configuração e rede dos computadores no período que estive na administração do LSD, aprendi muito neste período.
Agradeço também aos pesquisadores Amelia, Eduardo, Eduardo Devidson, Fernando Amaro, Jéssica, Josenilson, Lianna, Mauro, Raimundo Neto, Vladimir e todos os integrantes do Laboratório do grande professor Zair, pela amizade e pela força durante o mestrado.
Agradeço ao professor Zair, professor que admiro muito, pelo incentivo durante o mestrado e pelas aulas e lição de vida dentro e fora da sala de aula.
Agradeço muito a Cleonilson Protásio de Souza, Omar Andres Carmona Cortes e Rafael Fernandes Lopes pelo puxão de orelhas durante a graduação no IFMA e por terem me direcionado para este caminho.
A todos os grandes amigos e companheiros de pesquisa do IFMA em especial a Danielle Pinheiro, Fernanda, Jefferson Amaral, Paulo César, Rodrigo Miranda, Ricardo Johannsen e Thiago Nasper por sempre serem um amparo nos momentos difíceis e uma companhia nos momentos de descontração.
Agradeço também a Marcos e Anderson da Empresa EPS Tecnologia pela apoio quando decidi fazer o mestrado.
A todos os amigos que colecionei no IFMA e na UFMA. E a todos que de qualquer forma me ajudaram em minha formação e estiveram presentes quando precisei.
Sou muito grato também a Fundação de Amparo à Pesquisa e ao Desenvolvimento Científico e Tecnológico do Maranhão (FAPEMA) por ter financiado minha pesquisa durante todo o mestrado.
Agradeço à UFMA, Alcides e a todos do Programa de Pós-Graduação de Engenharia de Eletricidade.

"O que você sabe não tem valor, o valor está no que você faz com o que sabe."

Bruce Lee

Lista de Figuras

2.1 Taxonomia dos sistemas de grade 24

2.2 Componentes de um aglomerado InteGrade 29

3.1 Arquitetura básica do escalonador de aplicações em grades de computadores 33

3.2 Uma taxonomia hierárquica para algoritmos de escalonamento 34

3.3 Heurística Min-min 40

3.4 Heurística *XSufferage* 45

3.5 Heurística *Workqueue* com Replicação 47

3.6 Heurística *Storage Affinity* 49

3.7 Heurística RUMR 51

4.1 Mapeamento *SOFT* 66

4.2 Mapeamento *NICE* 67

5.1 Arquitetura do AGST (Autonomic Grid Simulation Tool) 70

5.2 Componentes do modelo de simulação. 75

5.3 Diagrama de Classes dos Monitores. 76

5.4 Classes que implementam o Mecanismo de Predição. 77

5.5 Diagrama de Classe da Heurística de Escalonamento. 80

5.6 1 aplicação a cada 0,002 segundos (0,12 min), fator de heterogeneidade igual a 3. 84

5.7 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 3. 84

5.8 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 6. 85

5.9 1 aplicação a cada 0,002 segundos (0,12 min), fator de heterogeneidade igual a 3. 87

5.10 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 3. 87

5.11 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 6. 88

Lista de Tabelas

2.1 Cinco maiores classes de aplicações de grades 23

5.1 Resumo dos parâmetros de simulação 83

Lista de Siglas

AGST *Autonomic Grid Simulator Tool*.

API *Application Programing Interface*.

BoT *Bag-of-Task*.

BSP *Bulk Synchronous Parallel*.

CPU *Central Processing Unit*.

CT *Completion Time*.

DAG *Directed Acyclic Graph*.

DBC *Deadline and Budget Constrained*.

DVS *Dynamic Voltage Scaling*.

EC Estimador de Custo.

EG Escalonador da Grade.

FCSA Ferramenta de Controle e Submissão de Aplicações.

FPLTF *Fastest Processor to Largest Task First*.

FTA *Failure Trace Archive*.

GB *Gigabyte*.

GHz *Gigahertz*.

GRL Gerenciador de Recursos Locais.

GRM *Global Resource Manager*.

GUPA *Global Usage Pattern Analyser*.

GWF *Grid Workload Format*.

IBM *International Business Machines*.

JVM *Java Virtual Machine*.

LRM *Local Resource Manager*.

LUPA *Local Usage Pattern Analyser*.

MAPE-K *Monitoring, Analysis, Planning, Execution and Knowledge*.

Mbps *Megabit per second*.

MCT *Minimum Conclusion Time*.

MI Milhares de Instruções.

MIPS Milhares de Instruções por Segundo.

MPI *Message Passing Interface*.

MPP *Massively Parallel Processing*.

MTBF *Mean Time Between Failures*.

MTTR *Mean Time To Recovery*.

OGST *Opportunistic Grid Simulator Tool*.

OVs Organizações Virtuais.

P2P *Peer-to-Peer*.

PVM *Parallel Virtual Machine*.

QoS *Quality of Service*.

RAM *Random Access Memory*.

SGBD Sistema Gerenciador de Banco de Dados.

SIG Serviço de Informação da Grade.

SMP *Symmetric Multi-Processing*.

SWF *Standart Workload Format*.

TBA *Time to Become Available*.

WQR *Workqueue* com Replicação.

Sumário

Lista de Figuras **ix**

Lista de Tabelas **xi**

Lista de Siglas **xii**

1 Introdução **18**

1.1 Objetivos 19

1.2 Estrutura da Dissertação 20

2 Grades de Computadores **21**

2.1 Introdução a Grades de Computadores 21

2.2 Taxonomia dos Sistemas de Grade 23

2.3 Grades de Computadores Pessoais (*Desktop Grids*) Oportunistas 25

2.3.1 Tipo de Aplicações em Grades de Computadores Oportunistas 26

2.3.2 O *Middleware* InteGrade 27

2.4 Conclusões 30

3 Introdução ao Escalonamento em Grades de Computadores **31**

3.1 Visão Geral do Processo de Escalonamento de Aplicações em Grades de Computadores 31

3.1.1 Arquitetura Básica do Escalonador de Aplicações em Grades de Computadores 32

3.2 Taxonomia dos Algoritmos de Escalonamento 34

3.2.1 Local vs. Global 35

3.2.2 Estático vs. Dinâmico 35

3.2.3 Ótimo vs. Sub-ótimo . . . 36
3.2.4 Algoritmos de Aproximação vs. Heurísticas . . . 36
3.2.5 Distribuídos vs. Centralizados . . . 37
3.2.6 Cooperativo vs. Não Cooperativo . . . 37
3.3 Algoritmos de Escalonamento para Grades de Computadores . . . 38
3.3.1 MCT . . . 39
3.3.2 Min-min . . . 39
3.3.3 *Task Grouping* . . . 41
3.3.4 *Dynamic FPLTF* . . . 41
3.3.5 *Sufferage* . . . 43
3.3.6 *XSufferage* . . . 43
3.3.7 *Workqueue* . . . 45
3.3.8 *Workqueue* com Replicação . . . 46
3.3.9 *Storage Affinity* . . . 47
3.3.10 RUMR . . . 49
3.4 Escalonamento de Aplicações de Acordo com Requisitos de QoS Especificados . . . 52
3.5 Conclusões . . . 56

4 Abordagem Proposta **57**

4.1 Arquitetura da Abordagem Proposta . . . 57
4.1.1 Mecanismo de Predição . . . 59
4.1.2 Mecanismo Adaptativo de Tolerância a Falhas . . . 60
4.1.3 Mecanismos de Monitoramento . . . 63
4.1.4 Heurística de Mapeamento Proposta . . . 65
4.2 Conclusões . . . 68

5 Avaliação da Abordagem Proposta **69**

5.1 A Ferramenta AGST 69
5.1.1 Geração Automatizada de Aplicações e Tarefas 72
5.1.2 Geração Automatizada de Recursos Computacionais e Enlaces de Rede . 72
5.1.3 Falhas de Recursos e Técnicas de Tolerância a Falhas 73
5.1.4 Simulação de Comportamento Autonômico 73
5.2 Implementação do Modelo de Simulação da Abordagem Proposta 74
5.2.1 Mecanismos de Monitoramento 75
5.2.2 Mecanismo de Predição 77
5.2.3 Mecanismo de Tolerância a Falhas 78
5.2.4 Heurística de Mapeamento Proposta 79
5.3 Experimentos Realizados 81
5.3.1 Avaliação sem Falhas de Recursos 82
5.3.2 Avaliação com Falhas de Recursos 85
5.4 Conclusões 89

6 Conclusão e Trabalhos Futuros **91**

6.1 Trabalhos Futuros 92

Referências Bibliográficas **93**

1 Introdução

Uma grade de computadores é um sistema que coordena recursos distribuídos, utilizando protocolos e interfaces padrões de forma a permitir a integração e compartilhamento de recursos computacionais como, por exemplo, poder computacional, software, dados e periféricos em redes corporativas e entre instituições. A tecnologia de grades de computadores recebe hoje grande atenção por parte tanto da academia quanto da indústria por ter se firmado como uma alternativa atraente para a execução de uma larga variedade de aplicações que requeiram grande poder computacional ou que processem grandes volumes de dados nas mais variadas áreas como biologia computacional, previsão do tempo e simulações de mercado.

Atualmente, as instituições privadas e públicas possuem um grande número de recursos computacionais, tais como computadores pessoais e estações de trabalho, com grande capacidade de processamento e armazenamento de dados. Os computadores estão ociosos na maior parte do tempo e, mesmo quando estão em uso, normalmente apenas uma pequena percentagem de sua capacidade de computação é efetivamente utilizada [43] [46] [44]. Grades oportunistas são sistemas computacionais que proveem meios para usar uma base instalada de computadores para execução de aplicações computacionais de alto desempenho, aproveitando o poder de computação ocioso disponível [16]. Elas são usualmente construídas a partir de computadores pessoais não dedicados para a execução das aplicações submetidas à grade, que devem compartilhar os recursos computacionais dos nós onde são executadas com uma carga de trabalho gerada pelos seus usuários locais.

O processo de escalonamento de aplicações em um sistema de grade consiste na atribuição das tarefas das aplicações para os recursos disponíveis seguindo um objetivo específico, como o de minimizar o tempo de resposta das aplicações e/ou maximizar o uso dos recursos computacionais disponíveis. No entanto, a construção de uma estratégia de escalonamento voltada a ambientes de grades oportunistas é uma tarefa desafiadora devido a diversas características presentes nestes ambientes computacionais, tais como: (a) grande instabilidade, decorrente do fato dos nós não serem dedicados e as aplicações não executarem em um ambiente

controlado; (b) alta heterogeneidade dos recursos computacionais e dos enlaces de rede, usualmente compreendendo recursos computacionais espalhados por diferentes domínios administrativos; (c) necessidade do middleware da grade não interferir no uso regular dos recursos, o que requer migração e reescalonamento de aplicações sempre que recursos tornarem-se indisponíveis; (d) falhas de recursos, visto que o dinamismo de recursos da grade pode gerar falhas que não foram previstas para um experimento, o que também requer migração e reescalonamento de aplicações. Além disso, os recursos de uma grade oportunista podem ser utilizados a qualquer momento para a execução de tarefas locais, tornando difícil a previsão do término das tarefas em execução nos nós da grade. Em particular, estas características dificultam a execução de aplicações para as quais existem restrições de tempo para sua conclusão neste tipo de ambiente computacional.

1.1 Objetivos

Esta dissertação tem por objetivo a concepção de um mecanismo para o gerenciamento da execução de aplicações que possuam restrições para seu tempo de execução. Essas restrições são definidas pelos usuários no ato de sua submissão a um sistema de grade oportunista.

Os objetivos específicos deste trabalho são:

- Estudo e avaliação do estado da arte de grades de computadores e ambientes de grades oportunistas;
- Estudo e avaliação do estado da arte de algoritmos de escalonamento em grades computacionais;
- Estudo e avaliação do estado da arte de estratégias de escalonamento que visa obter uma melhor qualidade do serviço disponibilizado aos seus usuários;
- Projeto e desenvolvimento de um mecanismo de gerenciamento da execução de aplicações em grades oportunistas;
- Avaliação do mecanismo de gerenciamento da execução de aplicações proposto através do uso de técnicas de simulação.

1.2 Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta uma breve fundamentação sobre grades de computadores, onde serão apresentados conceitos, características, aplicações, desafios e uma taxonomia da computação em grade. Neste capítulo é também apresentado o conceito de grades oportunistas. O Capítulo 3 apresenta uma descrição sobre o processo de escalonamento em grades de computadores e uma taxonomia para os algoritmos de escalonamento. Neste capítulo são descritos algoritmos de escalonamento tipicamente empregados em grades de computadores, incluindo abordagens que gerenciam restrições de tempo para a execução de aplicações que serviram de inspiração para este trabalho. No Capítulo 4 descreve-se o mecanismo proposto neste trabalho, enquanto o Capítulo 5 apresenta os detalhes de sua implementação no simulador AGST e os resultados da avaliação da estratégia proposta. Por fim, no Capítulo 6 são descritas as conclusões obtidas a partir deste trabalho e perspectivas futuras que podem advir deste esforço inicial.

2 Grades de Computadores

Este capítulo apresenta uma breve fundamentação teórica sobre as grades de computadores. Serão apresentados conceitos, características, aplicações, desafios e uma taxonomia para a computação em grade. Neste capítulo são apresentados ainda as grades de computadores pessoais (*desktop grids*) oportunistas e suas características, dado que este trabalho está inserido no contexto de um projeto voltado para esta área.

2.1 Introdução a Grades de Computadores

Atualmente, diversos ramos de atividades científicas, comerciais e industriais como biologia, processamento de imagens para diagnóstico médico, previsão do tempo, física de alta energia, previsão de terremotos, simulações mercadológicas, prospecção de petróleo e computação gráfica, requerem grande capacidade de processamento, compartilhamento de dados e colaboração entre usuários e organizações. Porém, alternativas tradicionais para a realização de computação de alto desempenho, como o uso de supercomputadores ou de aglomerados de computadores como clusters Beowulf, requerem altos investimentos em hardware e software, podendo chegar a custar centenas de milhares de dólares.

Por outro lado, redes de computadores existentes em instituições tanto públicas quanto privadas formam hoje um enorme parque computacional interconectado principalmente por tecnologias ligadas à Internet. No entanto, apesar de permitir comunicação e troca de informações entre computadores, as tecnologias que compõem a Internet atual não disponibilizam abordagens integradas que permitam o uso coordenado de recursos pertencentes a várias instituições na realização de computações.

Uma abordagem, denominada Computação em Grade (*Grid Computing*) [25] [5], foi desenvolvida com o objetivo de superar esta limitação. A origem do termo *Grid Computing* deriva de uma analogia com a rede elétrica (*Power Grid*), e reflete o objetivo

de tornar o uso de recursos computacionais distribuídos tão simples quanto ligar um aparelho na rede elétrica.

A computação em grade permite a integração e o compartilhamento de computadores e recursos computacionais, como software, dados e periféricos, em redes corporativas e entre estas redes, estimulando a cooperação entre usuários e organizações, criando ambientes dinâmicos e multi-institucionais, fornecendo e utilizando os recursos de maneira a atingir objetivos comuns e individuais [3] [26].

Os primeiros projetos de pesquisa na área de computação em grade surgiram nos anos 90 com o objetivo de interligar centros de supercomputação. A abordagem na época era conhecida como metacomputação [4]. Projetos como FAFNER[1], SETI@home[2] e I-WAY [23] obtiveram grande repercussão e seus resultados serviram de base para o desenvolvimento de uma infraestrutura de grade mais ubíqua, capaz de interligar uma grande quantidade de instituições e recursos computacionais. Globus [24], Legion [31], OurGrid [2], GridBus [7] e InteGrade [16] são exemplos de projetos bem sucedidos desta segunda geração de grades de computadores.

O *middleware*[3] de grade é o componente de software central de um sistema de grade, sendo responsável pela integração dos recursos distribuídos, de modo a criar um ambiente unificado para o compartilhamento de dados, recursos computacionais e execução de aplicações.

O compartilhamento multi-institucional e dinâmico proposto pela computação em grade deve ser, necessariamente, altamente controlado, com provedores de recursos e consumidores definindo claramente e cuidadosamente o que é compartilhado, a quem é permitido compartilhar, e as condições sob as quais ocorre o compartilhamento. Ian Foster et al. [26], conceituam os grupos formados por indivíduos e/ou organizações que compõem esses ambientes dinâmicos e multi-institucionais como Organizações Virtuais (OVs).

[1] http://www.npac.syr.edy/factoring.html

[2] http://setiathome.ssl.berkeley.edu

[3] Middleware é uma camada de software que reside entre o sistema operacional e a aplicação a fim de facilitar o desenvolvimento de aplicações, escondendo do programador diferenças entre plataformas de hardware, sistemas operacionais, bibliotecas de comunicação, protocolos de comunicação, formatação de dados, linguagens de programação e modelos de programação.

Existem atualmente muitas aplicações para a computação em grade. Ian Foster e Carl Kesselman [25] identificaram as cinco principais classes de aplicações para grades de computadores: supercomputação distribuída, alto rendimento, computação sob demanda, computação intensiva de dados e computação colaborativa. Estas classes de aplicações são apresentadas na Tabela 2.1.

Categoria	**Características**	**Exemplos**
Supercomputação distribuída	Grandes problemas com intensiva necessidade de CPU, memória, etc.	Simulações interativas distribuídas, cosmologia, modelagem climática
Alto rendimento	Agregar recursos ociosos para aumentar a capacidade de processamento. A grade é utilizada para executar uma grande quantidade de tarefas independentes ou fracamente acopladas.	Projeto de chips, problemas de criptografia, simulações moleculares
Computação sob demanda	Recursos remotos integrados em computações locais, muitas vezes por um período de tempo limitado	Instrumentação médica, processamento de imagens microscópicas
Computação intensiva de dados	Síntese de novas informações de muitas ou grandes fontes de dados	Experimentos de alta energia, modernos sistemas meteorológicos
Computação colaborativa	Suporte à comunicação ou a trabalhos colaborativos entre vários participantes	Educação, projetos colaborativos

Tabela 2.1: Cinco maiores classes de aplicações de grades

2.2 Taxonomia dos Sistemas de Grade

Krauter et al. [38] propõem uma taxonomia para sistemas de grades de computadores ilustrada na Figura 2.1 e descrita a seguir.

- **Grade Computacional** (*Computing Grid*): é um sistema de computação que objetiva prover maior capacidade computacional através da agregação de recursos computacionais distribuídos. Como exemplo de classes de aplicações que utilizam este tipo de grade temos: supercomputação distribuída, cujas aplicações usam grades para agregar recursos computacionais substanciais para

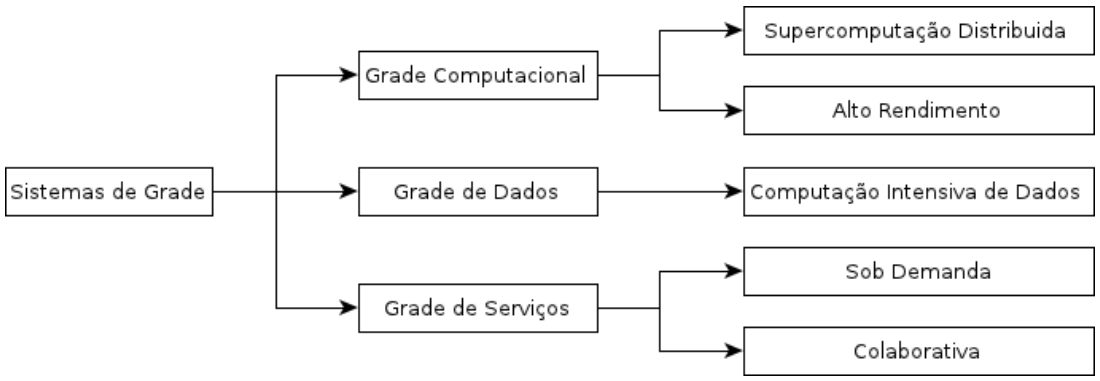

Figura 2.1: Taxonomia dos sistemas de grade

resolver problemas que não poderiam ser resolvidos por um único sistema como, por exemplo, aplicações para planejamento e treinamento militar através de simulações interativas distribuídas e simulação precisa de processos físicos complexos; modelagem climática; aplicações de alto rendimento, onde a grade é usada para escalonar grande número de tarefas independentes ou fracamente acopladas, com o objetivo de utilizar ciclos de processadores ociosos como, por exemplo, a resolução de problemas criptográficos;

- **Grade de Dados** (*Data Grid*): é um sistema de computação em grade que objetiva prover mecanismos especializados para publicação, descoberta, acesso e classificação de grandes volumes de dados distribuídos. A computação intensiva de dados cujo foco está na síntese de novas informações de dados que são mantidos em repositórios, bibliotecas digitais e bancos de dados geograficamente distribuídos são exemplos de grades de dados.

- **Grade de Serviços** (*Service Grid*): provê serviços viabilizados pela integração de diversos recursos computacionais. Exemplos de classes de aplicações são:

  - **Computação sob demanda:** aplicações usam as capacidades da grade por períodos curtos de acordo com solicitações por recursos como, por exemplo, aplicações de instrumentação médica e requisições de utilização de software especializado por usuários remotos;
  - **Computação colaborativa:** aplicações que utilizam a grade para dar apoio a trabalhos cooperativos envolvendo diversos participantes, como, por exemplo, em projetos colaborativos e educação;
  - **Multimídia:** a grade provê a infra-estrutura para aplicações multimídia em tempo real.

2.3 Grades de Computadores Pessoais (*Desktop Grids*) Oportunistas

Atualmente, as instituições privadas e públicas têm um grande número de recursos de computação, tais como computadores pessoais e estações de trabalho, com grande capacidade de processamento e armazenamento de dados. Os computadores estão ociosos na maior parte do tempo e, mesmo quando estão em uso, normalmente apenas uma pequena percentagem de sua capacidade de computação é efetivamente utilizada [43, 44, 46]. Grades oportunistas são sistemas computacionais que provêem meios para usar uma base instalada de computadores pessoais para execução de aplicações computacionais de alto desempenho, aproveitando o poder de computação ocioso disponível [28].

Uma abordagem possível para estruturar uma grade é o uso oportunista de estações de trabalho regulares de usuários. O foco de um middleware de grade oportunista não é a integração de clusters de computadores dedicados (ex.: Beowulf) ou recursos de supercomputação, mas no aproveitamento de ciclos de computação ocioso de computadores pessoais e estações regulares de trabalho que podem ser distribuídos por vários domínios administrativos.

Em um *desktop grid*, um grande número de computadores pessoais regulares são integrados para a execução de aplicações distribuídas em larga escala. Os recursos computacionais são heterogêneos em relação à sua configuração de hardware e software. Várias tecnologias de rede podem ser usadas na interconexão de redes, resultando em enlaces com diferentes capacidades no que diz respeito a propriedades como largura de banda, taxa de erros e latência de comunicação. Os recursos de computação também podem ser distribuídos por vários domínios administrativos. No entanto, do ponto de vista do usuário, o sistema de computação deve ser visto como um recurso único e integrado, devendo ainda ser fácil de usar.

Se o *middleware* de grade segue uma abordagem oportunista, os recursos não precisam ser dedicados para a execução de aplicações da grade. A carga de trabalho da grade irá coexistir com pedidos de execução de aplicações locais apresentados pelos usuários das máquinas regulares. O *middleware* da grade deve aproveitar ciclos ociosos de computação que surgem de períodos de tempo sem uso das estações de trabalho que compõem a grade.

Ao aproveitar o poder computacional ocioso de estações de trabalho existentes e conectá-los a uma infra-estrutura de rede, o *middleware* da grade permite uma melhor utilização dos recursos computacionais existentes e permite a execução de aplicações paralelas computacionalmente intensivas que de outro modo exigiria um aglomerado ou máquinas paralelas de alto custo.

2.3.1 Tipo de Aplicações em Grades de Computadores Oportunistas

Durante a última década, desenvolvedores de *middleware* para *desktop grids* oportunistas tem trabalhado na construção de várias abordagens para permitir a execução de diferentes classes de aplicações, tais como:

- **Aplicações Regulares ou sequenciais**, neste tipo de aplicação, o código executável é designado para um único nó da grade.
- **Aplicações paralelas fracamente acopladas (paramétricas ou *Bag-of-Task* (BoT))**, neste tipo de aplicação, várias cópias do código executável são designadas para diferentes nós da grade, sendo que cada tarefa possui um subconjunto dos dados de entrada que são processados de forma independente e sem a troca de dados. Neste caso, não existe comunicação entre os nós, pois as tarefas são independentes entre si e podem ser executadas em qualquer ordem. Aplicações que fazem simulação, mineração de dados e renderização de imagem são exemplos de aplicações desse tipo.
- **Aplicações paralelas fortemente acopladas** que seguem o modelo *Bulk Synchronous Parallel* (BSP) [51] ou o padrão *Message Passing Interface* (MPI), neste tipo de aplicação, vários nós são designados para a execução da aplicação e as computações são realizadas em cada um dos nós participantes mas, neste caso, os processos ocasionalmente trocam dados entre si através da troca de mensagens ou abstrações de memória compartilhada.
- **Aplicações *Parameter Sweep* (varredura de parâmetros)**, neste tipo de aplicação, o mesmo código é executado várias vezes usando diferentes valores de entrada, tendo, assim, diferentes valores de saída. Exemplos de aplicações *Parameter Sweep* são simulações de Monte Carlo.

2.3.2 O *Middleware* InteGrade

O projeto InteGrade[4] [16, 28] é uma iniciativa multi-institucional da qual a Universidade Federal do Maranhão fez parte para o desenvolvimento de um *middleware* de grade capaz de usufruir do poder computacional de estações de trabalho que estejam ociosas, aproveitando seus ciclos para a execução de aplicações paralelas de alto desempenho. O objetivo é permitir que organizações utilizem sua infraestrutura computacional para realizar este tipo de computação, sem a necessidade de adquirir novo e específico hardware.

Devido à heterogeneidade, alta escalabilidade e dinamismo do ambiente de execução, prover um suporte eficiente para a execução de aplicações em grades oportunistas compreende um grande desafio para os desenvolvedores de *middleware*, que devem prover soluções inovadoras para resolver os problemas encontrados em áreas, tais como:

1. **Gestão de recursos**: que engloba desafios como a monitoração eficiente de um grande número de recursos de computação distribuídos pertencentes a múltiplos domínios administrativos. Em grades oportunistas, esta questão é ainda mais difícil devido à natureza dinâmica do ambiente de execução, onde os nós podem entrar e sair da rede a qualquer momento, devido à utilização de máquinas não-dedicadas por seus usuários normais locais.

2. **Programação de aplicações e gerenciamento de execução**: que deve fornecer mecanismos amigáveis para executar aplicações no ambiente de grade, para controlar a execução das tarefas e fornecer ferramentas para coletar os resultados da execução de aplicações e gerar relatórios sobre as situações atuais e passadas. O gerenciamento de execução da aplicação deve abranger todos os modelos de execução suportados pelo *middleware*.

3. **Tolerância a falhas**: ambientes de grade são altamente propensos a falhas. Esta característica é amplificada em grades oportunistas devido ao seu dinamismo e a utilização de máquinas não-dedicadas, levando a um ambiente de computação não-controlado. Um mecanismo de detecção eficiente e escalável de falhas deve ser fornecido pelo *middleware* de grade, juntamente com um mecanismo

[4]Página Inicial: `http://www.integrade.org.br/`

de recuperação automática da execução de aplicações, que não requeira a necessidade de intervenção humana.

Mesmo em face aos desafios encontrados, o *middleware* InteGrade foi construído como um sistema orientado a objetos que provê uma infra-estrutura de *software* robusta e flexível para computação em grade oportunista, com suporte de aplicações paralelas fortemente acopladas. O InteGrade é estruturado em aglomerados, ou seja, unidades autônomas dentro da grade que contém todos os componentes necessários para funcionar independentemente. Estes aglomerados podem estar organizados de uma forma hierárquica ou conectados através de uma rede ponto-a-ponto (*Peer-to-Peer* (P2P)) [27].

Cada aglomerado contém um nó gerenciador do aglomerado (*Cluster Manager*) que executa componentes do InteGrade responsáveis pelo gerenciamento de recursos computacionais e pela comunicação entre aglomerados. Os outros nós do aglomerado que são chamados de *Resource Providers* são estações de trabalho que exportam parte dos seus recursos para usuários da grade. Os nós que provêem recursos podem ser máquinas compartilhadas ou dedicadas. Cada aglomerado do InteGrade possui um escalonador central, responsável por realizar o escalonamento das aplicações nos recursos locais a este aglomerado. No entanto, caso não haja recursos suficientes para realizar o escalonamento de um conjunto de tarefas pertencentes a uma mesma aplicação paralela, o escalonador encaminha a aplicação para que seja executada em outro aglomerado da hierarquia.

Atualmente o InteGrade permite a execução de aplicações regulares, paramétricas e paralelas. O usuário do InteGrade pode, opcionalmente, especificar os requisitos indispensáveis da aplicação como, por exemplo, a arquitetura para a qual a aplicação foi compilada e as preferências que guiam a ordenação para a escolha dos recursos candidatos (e.g. recursos que possuem a maior quantidade de memória livre). De posse dessas informações sobre as exigências da aplicação, o escalonador do aglomerado seleciona e ordena um conjunto de recursos candidatos. As tarefas das aplicações são então escalonadas para os recursos de acordo com a lista de recursos ordenados.

A arquitetura do InteGrade dispõe de vários componentes que permitem a execução da aplicação, estes componentes podem ser vistos na Figura 2.2.

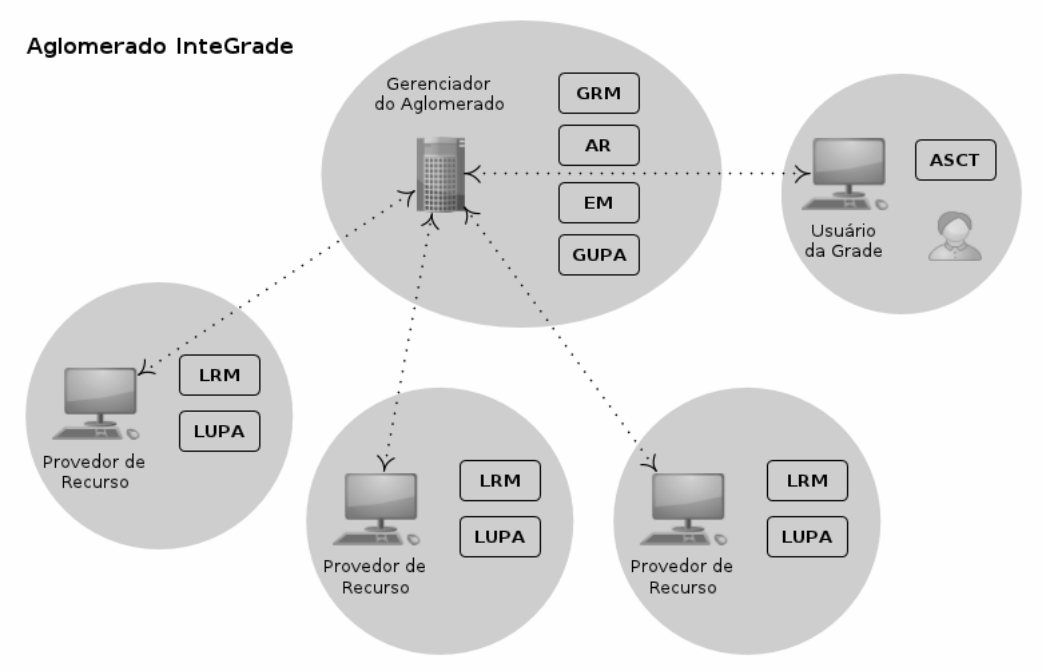

Figura 2.2: Componentes de um aglomerado InteGrade

- **AR** (*Application Repository*): armazena binários de aplicações submetidas por usuários para execução na grade. Toda aplicação submetida à execução deve ser primeiramente registrada neste repositório.

- **ASCT** (*Application Submission and Control Tool*): ferramenta para submissão de aplicações para execução na grade. Os usuários podem especificar pré-requisitos de execução, como a plataforma de *hardware* e *software* nas quais a aplicação deve ser executada, requisitos de memória necessários à execução da aplicação, entre outros. Também é possível monitorar o progresso da aplicação em execução.

- **LRM** (*Local Resource Manager*): é executado em cada nó que fornece recursos ao aglomerado. O LRM coleta informações sobre o estado do nó como memória, *Central Processing Unit* (CPU), disco e utilização da rede, enviando estas informações periodicamente ao GRM de seu aglomerado, através de um "protocolo de atualização de informações".

- **GRM** (*Global Resource Manager*): gerencia os recursos computacionais constantes de um aglomerado de máquinas conectadas por uma rede local e provê mecanismos de escalonamento baseado no conhecimento que ele possui sobre a disponibilidade dos recursos e os requisitos especificados pelo usuário no processo de submissão para execução de aplicações. O armazenamento e

consulta das informações de estado dos recursos (LRMs) são realizados pelo serviço de negociação (*Trading*) definido em CORBA [45].

- **EM** (*Execution Manager*): mantém um banco de dados com todas as aplicações que foram submetidas para a execução no aglomerado, incluindo a localização de cada processo da aplicação e o estado da requisição de execução. Ele também coordena o processo de recuperação das aplicações em caso de falhas.

- **LUPA** (*Local Usage Pattern Analyser*): executa junto ao LRM e coleta localmente informações de padrões de uso dos usuários do nó. Baseia-se em longas séries de dados derivadas de padrões semanais de uso do nó. Periodicamente transmite as informações locais ao GUPA.

- **GUPA** (*Global Usage Pattern Analyser*): gerencia as informações sobre os padrões de uso de recursos dos nós componentes do aglomerado. Fornece estas informações ao GRM de forma a colaborar para que ele tome melhores decisões de escalonamento.

2.4 Conclusões

Este capítulo apresentou um introdução à tecnologia de grades de computadores. Foram descritas as classes de aplicações que fazem uso deste tipo de tecnologia e uma taxonomia que classifica as grades de computadores em três tipos: computacionais, de dados e de serviços. Por fim, foi abordado o contexto de grade oportunistas, que permite o aproveitamento de recursos ociosos para o estabelecimento de grades de computadores, onde foram descritas os principais tipos de aplicações que podem ser executados na grade e foi visto o *Middleware* InteGrade.

3 Introdução ao Escalonamento em Grades de Computadores

Este capítulo apresenta uma visão geral do processo de escalonamento em grades de computadores. Descreve-se uma taxonomia básica para os algoritmos de escalonamento. Por fim, são apresentados vários algoritmos de escalonamento usualmente empregados em ambientes de grades computacionais, incluindo abordagens que visam gerenciar restrições de tempo impostas para a execução de aplicações.

3.1 Visão Geral do Processo de Escalonamento de Aplicações em Grades de Computadores

O escalonamento em grade é um processo de tomada de decisões envolvendo recursos pertencentes a múltiplos domínios administrativos. Este processo inclui a pesquisa de recursos para a execução de aplicações na grade. No entanto, diferentemente dos escalonadores tradicionais de sistemas distribuídos e paralelos (ex: *Massively Parallel Processing* (MPP) e *Symmetric Multi-Processing* (SMP)), os escalonadores de grades não possuem controle sobre os recursos e o conjunto de aplicações no sistema. Assim, torna-se importante a existência de componentes que permitam, entre outras funcionalidades, a descoberta de recursos, o monitoramento e armazenamento de informações sobre recursos e aplicações, a permissão de acesso a diferentes domínios administrativos e, dependendo da estratégia de escalonamento adotada, a estimativa do desempenho dos recursos e do tempo de execução das aplicações nos mesmos.

Ao longo deste trabalho, adota-se as seguintes definições relativas a termos utilizados com frequência ao se descrever o processo de escalonamento em grades de computadores:

- Uma tarefa é uma unidade atômica para ser escalonada pelo escalonador e atribuída a um recurso.

- Tarefas podem possuir propriedades tais como: exigências relativas a recursos, como uma arquitetura de CPU, limites mínimos de memória para sua execução, prazo para sua conclusão (*deadline*), prioridade, etc.

- Uma aplicação (ou *job*) é composta por um conjunto de tarefas atômicas que serão executadas em um conjunto de recursos.

- Um recurso é algo que é necessário para realizar uma operação como, por exemplo, um processador para execução de tarefas, um dispositivo de armazenamento de dados ou um enlace de rede para o transporte de dados.

- Um nó (*site* ou aglomerado) é uma entidade autônoma composta por um ou mais recursos.

- O processo de escalonamento de tarefas refere-se ao mapeamento (atribuição) de tarefas a nós que as executarão, que podem estar distribuídos em vários domínios administrativos.

3.1.1 Arquitetura Básica do Escalonador de Aplicações em Grades de Computadores

A arquitetura básica do escalonamento de aplicações em grades de computadores pode ser vista na Figura 3.1 que ilustra os passos básicos do processo de escalonamento em grades de computadores, estendido a partir da arquitetura exposta por Zhu [55]. Uma vez que os usuários submetem aplicações para a grade, através da Ferramenta de Controle e Submissão de Aplicações (FCSA) (passo 1), o processo de escalonamento em grades, segundo Schopf et al. [49], pode ser dividido em três estágios: filtro e descoberta de recursos, seleção dos recursos e envio da aplicação com preparação do ambiente de execução.

O Escalonador da Grade (EG), no primeiro estágio, cria um filtro para selecionar os recursos a partir das restrições e preferências providas pelos usuários. As restrições definem requisitos mínimos para a seleção de máquinas como, por exemplo, a utilização de máquinas que tenham pelo menos 4 *Gigabyte* (GB) de memória *Random Access Memory* (RAM). Já as preferências definem a ordem na escolha dos recursos como, por exemplo, ordenar as máquinas pela maior quantidade de memória disponível.

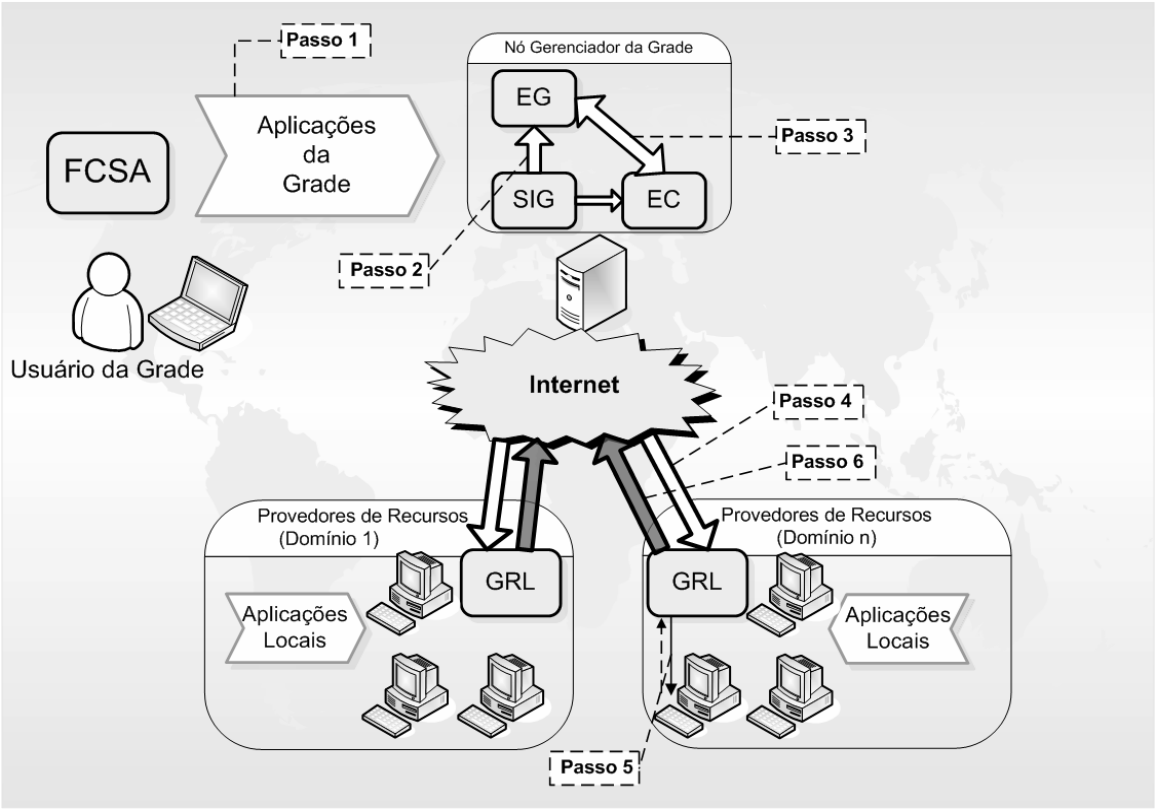

Figura 3.1: Arquitetura básica do escalonador de aplicações em grades de computadores

As informações sobre os recursos disponíveis são muito importantes para um escalonamento, especialmente devido à natureza heterogênea e dinâmica da grade. O papel do Serviço de Informação da Grade (SIG) é o de prover um conjunto de informações dinâmicas e estáticas para os escalonadores da grade. As informações dinâmicas de cada recurso são constantemente monitoradas pelo Gerenciador de Recursos Locais (GRL) e enviadas periodicamente para o SIG. Alguns exemplos de informações dinâmicas são: a capacidade livre de CPU, a quantidade da memória em uso e o atraso para entrega de pacotes nos enlaces da rede. Por outro lado, as informações estáticas obtidas pelo SIG são enviadas pelo GRL, geralmente, no momento de registro de cada recurso na grade. Alguns exemplos de informações estáticas são: a capacidade da CPU, o tamanho da memória de nós da grade e a configuração e capacidade dos enlaces de rede.

O Estimador de Custo (EC) provê uma medida de como um recurso computacional executa um determinado tipo de aplicação. Um *benchmark* analítico poderia ser usado como forma de ordenar os recursos disponíveis de acordo com a sua eficiência para executar um determinado tipo de código computacional.

Após filtrar os recursos, no segundo estágio do processo de escalonamento, o EG irá gerar o mapeamento das aplicações para os recursos de acordo com o objetivo do sistema como, por exemplo, minimizar o tempo de resposta das aplicações ou maximizar o número de aplicações concluídas por unidade de tempo (*throughput*) [55] [20]. Este mapeamento irá ocorrer de acordo com informações sobre o estado dos recursos disponíveis e a estimativa do desempenho de recursos para tipos específicos de aplicações fornecidas, respectivamente, pelo Serviço de Informação da Grade (SIG) (passo 2) e um Estimador de Custo (EC) (passo 3).

No terceiro estágio do processo de escalonamento, o GRL é responsável por receber as aplicações enviadas pelo Escalonador da Grade (EG) (passo 4) e preparar o ambiente para sua execução como, por exemplo, realizar a transferência de arquivos contendo dados de entrada para a aplicação. Após a preparação do ambiente, o GRL inicia a execução das aplicações (passo 5). Quando as aplicações terminam sua execução, ele envia os resultados da execução das aplicações para os clientes da grade (passo 6).

3.2 Taxonomia dos Algoritmos de Escalonamento

Casavant et al. [9], propõem uma taxonomia hierárquica para os algoritmos de escalonamento em sistemas paralelos de propósito geral e computação distribuída. A taxonomia pode ser vista na Figura 3.2, explicada a seguir.

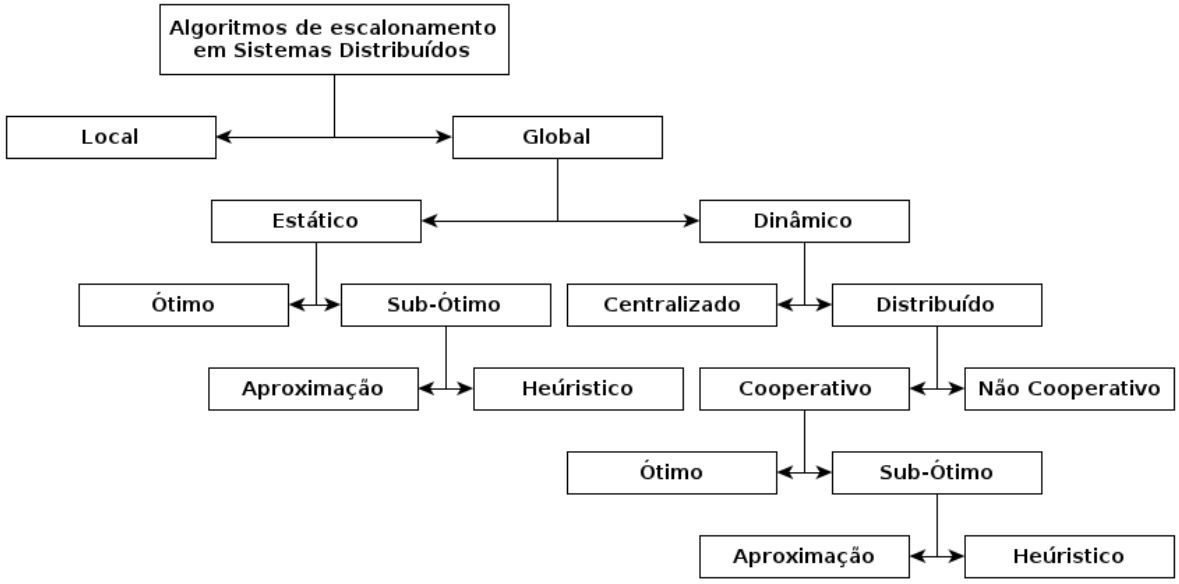

Figura 3.2: Uma taxonomia hierárquica para algoritmos de escalonamento

3.2.1 Local vs. Global

Escalonamento local está relacionado a atribuição de processos a fatias de tempo (*time-slices*) de um único processador. Escalonamento global é o problema de se decidir onde executar um processo em um ambiente multiprocessado, como um ambiente distribuído composto por diversas máquinas conectadas em uma rede. Escalonadores globais são responsáveis por escolher quando e quais processos terão acesso a quais recursos do sistema [34] [41]. Perceba que esta definição não implica que o escalonamento global seja realizado por uma única autoridade central.

3.2.2 Estático vs. Dinâmico

A escolha entre escalonamento estático e dinâmico está relacionada ao momento no qual decisões de escalonamento são realizadas. No escalonamento estático, cada imagem executável é estaticamente atribuída para execução em um processador particular, e toda vez que a imagem for submetida para execução ela é atribuída a este processador. Uma definição menos formal para o escalonamento estático inclui algoritmos que atribuem cada tarefa de uma aplicação somente uma vez para os recursos que as executarão. Portanto, a atribuição da aplicação é estática, e pressupõe que uma estimativa do custo da computação em cada nó pode ser realizada antes de sua execução. Um problema nesta abordagem, é que informações estáticas não permitem uma adaptação a situações como a falha de um nó, seu isolamento devido a falhas de rede, ou uma sobrecarga que torne seu tempo de resposta maior que o esperado.

No escalonamento dinâmico a alocação de tarefas pode ocorrer durante a execução da aplicação e leva em consideração informações de contexto dinâmicas do estado do ambiente, como o tempo médio entre falhas *Mean Time Between Failures* (MTBF) dos nós e a carga de trabalho dos mesmos, entre outras. Escalonadores dinâmicos usualmente possuem dois componentes centrais: um mecanismo para estimar o estado dos recursos da grade e um mecanismo para tomada de decisão.

3.2.3 Ótimo vs. Sub-ótimo

Caso todas as informações a respeito do estado do sistema bem como das necessidades dos processos com relação aos recursos a serem utilizados sejam conhecidas, uma atribuição ótima de processos a recursos pode ser realizada baseada em uma função de critério. Exemplos de medidas de otimização incluem a minimização do tempo total de conclusão dos processos, a maximização da utilização dos recursos do sistema ou a maximização da vazão (*throughput*) do sistema. Caso estes problemas sejam computacionalmente inviáveis (dado que o problema do escalonamento é NP-Completo [21, 32]), soluções sub-ótimas podem ser utilizadas. Duas categorias de algoritmos sub-ótimos são apresentadas a seguir: algoritmos de aproximação e heurísticas.

3.2.4 Algoritmos de Aproximação vs. Heurísticas

Algoritmos de aproximação utilizam o mesmo modelo computacional formal de algoritmos ótimos, mas ao invés de percorrer todo o espaço da solução a procura de uma solução ótima satisfazem-se ao encontrar uma "boa" solução. O problema de reconhecer uma "boa" solução pode não ser insignificante, mas em casos nos quais esteja disponível uma métrica para avaliar a solução esta técnica pode ser utilizada para diminuir o tempo necessário para encontrar uma solução aceitável (no caso, um escalonamento). Os fatores que determinam a viabilidade de se utilizar esta abordagem incluem:

- A disponibilidade de uma função para avaliar a solução.
- O tempo necessário para avaliar a solução.
- A capacidade de julgar o valor de uma solução ideal de acordo com algumas métricas.
- A disponibilização de um mecanismo inteligente para podar o espaço da solução.

O outro ramo na categoria sub-ótima é chamado de heurística. Este ramo representa a classe de algoritmos estáticos que se baseiam em suposições obtidas a partir de um conhecimento prévio sobre os processos e características da

carga do sistema. Escalonadores baseados em heurísticas utilizam parâmetros que afetam o sistema de diferentes modos. Usualmente, o parâmetro a ser monitorado é correlacionado ao desempenho do sistema de forma indireta. Por exemplo, o agrupamento de processos que se comunicam frequentemente em um mesmo processador e a separação física de processos que podem se beneficiar de paralelismo diretamente diminui a sobrecarga envolvida na passagem de informações entre processadores, reduzindo a interferência entre processos que podem ser executados sem a necessidade de se sincronizar uns com os outros. Isto resulta em um impacto no serviço disponibilizado ao usuário, mas não pode ser diretamente relacionado (de forma quantitativa) ao desempenho do sistema da forma que o usuário enxerga. Desta forma, acredita-se que a adoção destas ações pode melhorar o desempenho do sistema, mas pode não ser possível provar uma relação de primeira ordem entre o mecanismo empregado e os resultados desejados.

3.2.5 Distribuídos vs. Centralizados

Em cenários de escalonamento dinâmico, a responsabilidade pela tomada de decisões de escalonamento global pode estar com um escalonador centralizado, ou ser compartilhado por vários escalonadores distribuídos. Assim, algoritmos de escalonamento centralizados são aqueles nos quais seu processamento é realizado em um único processador enquanto que algoritmos de escalonamento distribuído são aqueles nos quais o processo de escalonamento é distribuído fisicamente em mais de um processador.

3.2.6 Cooperativo vs. Não Cooperativo

Algoritmos de escalonamento distribuídos podem trabalhar de forma cooperativa ou de forma independente (não-cooperativa). No caso dos não-cooperativos, escalonadores individuais agem sozinhos como entidades autônomas e determinam como seus próprios recursos devem ser utilizados, independente de como suas decisões afetam o resto do sistema. No caso cooperativo, cada escalonador da grade tem a responsabilidade de realizar sua própria porção da tarefa de escalonamento, mas todos os escalonadores trabalham em conjunto para atingir um objetivo comum a todo o sistema.

Além dos aspectos explicitados pela taxonomia hierárquica apresentada, outras propriedades importantes dos algoritmos de escalonamento podem ser utilizadas como critério de classificação, que não são abrangidas por este método [20]. Por exemplo, um aspecto importante a ser levado em consideração para um algoritmo de escalonamento é se a abordagem leva ou não em consideração requisitos de *Quality of Service* (QoS), tais como o tempo que o usuário está disposto a esperar até a conclusão da aplicação (*deadline*) e o preço que o usuário está disposto a pagar pela realização desta computação (*budget*).

3.3 Algoritmos de Escalonamento para Grades de Computadores

Um algoritmo de escalonamento determina como as aplicações devem ser escalonadas e como os recursos devem ser utilizados, de acordo com as metas de desempenho da grade e as informações disponibilizadas. No entanto, o mapeamento de um conjunto de tarefas em um conjunto de recursos heterogêneos é um problema NP-completo bem conhecido [21,32].

Os algoritmos de escalonamento podem ser agrupados em dois modos: modo *batch* e modo *on-line* [41]. O modo de escalonamento *on-line* define que uma tarefa será mapeada para uma máquina assim que ela chega ao escalonador. Já no modo *batch*, as tarefas não são mapeadas assim que elas chegam. Ao invés disso, elas são colocadas dentro de um conjunto de tarefas, chamado de meta-tarefa e são mapeadas em tempos pré-escalonados, chamados eventos de mapeamento. Algoritmos *batch* comparam os requisitos de recursos das tarefas entre si para tomar uma decisão de mapeamento.

Os algoritmos de escalonamento trabalham de varias maneiras, por exemplo, alguns algoritmos de escalonamento necessitam conhecer o tempo que as aplicações levariam para ser executadas nos recursos da grade. Esta informação pode ser estimada através de algoritmos de predição [39] que seguem basicamente duas abordagens. Na primeira abordagem, calcula-se a estimativa do tempo de execução da aplicação baseado no registro de execuções anteriores da mesma ou de aplicações semelhantes. A segunda abordagem é baseada no conhecimento do modelo de execução da aplicação, usualmente aplicações paralelas com cargas de

trabalho divisíveis (ex: MPI ou *Parallel Virtual Machine* (PVM)). O código da aplicação é analisado, estimando-se o tempo de execução de cada tarefa de acordo com a capacidade dos recursos da grade.

A seguir são descritos algoritmos de escalonamento usualmente empregados em grades de computadores que objetivam diversas metas de desempenho, tais como: (a) aumentar a vazão (*throughtput*) do sistema: que é a medida feita a partir do número de processos finalizados por unidade de tempo; (b) diminuir o tempo de resposta, medida definida pela diferença entre o momento de término da execução da tarefa e seu instante de chegada na fila de processos, ou seja, essa medida é soma dos tempos gastos em fila de espera por recursos e na execução propriamente dita dos processos; (c) aumentar a utilização de recursos: o escalonador pode fazer com que os recursos do sistema, tais como CPU, memória ou rede, sejam utilizados ao máximo, mesmo que para atingir tal meta seja necessário esquecer outros critérios; e (d) balancear a carga do sistema, que consiste em não subutilizar recursos enquanto outros estão trabalhando em sua capacidade máxima, onde a intenção é distribuir os processos para os recursos de acordo com a capacidade dos mesmos. Nenhum dos algoritmos vistos nesta Seção leva em consideração os requisitos impostos pelos usuários. Estas abordagens são descritas na Seção 3.4.

3.3.1 MCT

O *Minimum Conclusion Time* (MCT) é uma heurística *on-line* que atribui cada tarefa para a máquina com o menor tempo de conclusão esperado para realizá-la. O tempo de conclusão corresponde à soma do tempo que levará para a máquina estar disponível para executar a tarefa, mais o tempo que a tarefa demora para ser executada. Sendo m a quantidade de recursos da grade, a complexidade de mapeamento é $O(m)$, dado que assim que uma tarefa chega todas as máquinas na grade são examinadas para determinar a máquina que possui o menor tempo de conclusão para a mesma.

3.3.2 Min-min

Min-min é uma heurística do tipo *batch* baseada no MCT, cujo pseudo-código é apresentado na Figura 3.3. O Min-min recebe como parâmetros de entrada um conjunto de tarefas não mapeadas $taskList$ e um conjunto de máquinas $resourceList$

constantes da grade. Ao iniciar, o algoritmo calcula o tempo de conclusão que cada tarefa na $taskList$ levaria para ser concluída em cada máquina constante na $resourceList$ (linhas de 2 até 6).

A seguir, o algoritmo procura o menor tempo de conclusão para cada tarefa obtendo, também, a máquina que alcançaria este valor (linhas de 9 a 11). O Min-min, então, procura a tarefa $minTask$ com o tempo de conclusão mínimo entre todas as tarefas da $taskList$ e a atribui para a máquina $minResource$ cuja execução deste tempo seria obtido (linhas 12 e 13). A tarefa mapeada é removida da $taskList$ (linha 14). Uma vez que o mapeamento da tarefa $minTask$ para a máquina $minResource$ incrementa o tempo que esta máquina ficará ocupada (linha 15), isto requer a atualização do tempo de conclusão da $resourceList$ para o restante das tarefas na $taskList$ (linha 16). As operações das linhas 8 até 17 são repetidas enquanto houver alguma tarefa para ser mapeada na grade.

```
1:  function schedule(Vector<Resource> resourceList, Vector<Task> taskList)
2:    for Task task : taskList do
3:      for Resource resource : resourceList do
4:        calculateCompletionTime(resource, task)
5:      end for
6:    end for
7:
8:    while taskList.hasTaskNotScheduled() do
9:      for Task task : taskList do
10:       Resource minResource = searchMinimumConclusionTime(task, resourceList)
11:     end for
12:     Task minTask = searchMinTask(taskList)
13:     assign(minTask, minResource)
14:     taskList.remove(minTask)
15:     update(resourceList)
16:     update(taskList)
17:   end while
18: end function
```

Figura 3.3: Heurística Min-min

Sendo m a quantidade de tarefas na $taskList$ e r a quantidade de recursos constante na $resourceList$, o cálculo do tempo de conclusão de cada tarefa em cada máquina constante na $resourceList$ (linhas 2 a 6) custa $O(mr)$. O laço nas linhas 8 a 17 é repetido m vezes e cada interação custa $O(mr)$. Logo, o custo total do algoritmo é $O(m^2r)$.

3.3.3 *Task Grouping*

O algoritmo *Task Grouping* é voltado para o escalonamento de aplicações constituídas por uma grande quantidade de tarefas de curta duração. Neste caso, o escalonamento e distribuição de cada tarefa individualmente levaria a uma sobrecarga do sistema durante a transmissão das tarefas para os recursos da grade. O algoritmo agrupa as tarefas da aplicação de acordo com seus tamanhos de computação (medido em milhões de instruções) e a capacidade de processamento dos recursos da grade. Cada grupo é enviado a um único recurso, reduzindo-se a carga de transmissão das tarefas.

3.3.4 *Dynamic FPLTF*

A heurística *Dynamic Fastest Processor to Largest Task First* (FPLTF) (*Dynamic Fastest Processor to Large Task First*) é uma variação da heurística FPLTF que é uma heurística para aplicações de uso intensivo de CPU (CPU bound). O FPLTF é um bom representante de escalonamento estático. Contudo, dado a dinamicidade e heterogeneidade de recursos presentes em grades de computadores, usualmente não são adotados escalonadores estáticos neste tipo de ambiente computacional [1]. Para lidar com esta dinamicidade típica de grades de computadores, foi desenvolvido o *Dynamic* FPLTF [15] que, como próprio nome indica, é um algoritmo dinâmico e heurístico. O *Dynamic* FPLTF é uma heurística do tipo *batch* cujo objetivo funcional é centrado no escalonamento de aplicações paramétricas (*Bag-of-Task* (Bot)).

A ideia básica do *Dynamic* FPLTF é alocar tarefas grandes aos nós que oferecerem o menor tempo de execução, objetivando otimizar o tempo de conclusão (*Completion Time* (CT)) das mesmas.

O *Dynamic* FPLTF requer três tipos de informações sobre o ambiente de execução para escalonar as tarefas [1] [14] [15] [35]: o tamanho da tarefa (*Task Size*); a carga da máquina (*Host Load*); e a velocidade da máquina (*Host Speed*). Estas informações são utilizadas para a priorizar tarefas mais longas, alocando-as para máquinas mais rápidas.

A Velocidade da máquina é expressa no algoritmo de forma relativa. Por exemplo, uma máquina que tem uma $VelocidadeDaMaquina = 2$ executa uma tarefa

duas vezes mais rápido que uma máquina com $VelocidadeDaMaquina = 1$. A Carga da máquina representa a fração da máquina que não está disponível para a aplicação. O Tamanho da tarefa é o tempo necessário para uma máquina com $VelocidadeDaMaquina = 1$ completar a tarefa quando $CargaDaMaquina = 0$.

No início do algoritmo, o tempo para ser tornar disponível (*Time to Become Available* (TBA)) de cada máquina é inicializado com 0, e as tarefas são ordenadas pelo tamanho (ordem decrescente). Portanto, a tarefa mais longa é alocada primeiro. Uma tarefa é alocada a uma máquina que fornece o menor tempo de conclusão, onde [14] [15] [35]: $CT = TBA + CustoDaTarefa$. O $CustoDaTarefa$ é calculado da seguinte forma: $CustoDaTarefa = (TamanhoDaTarefa/VelocidadeDaMaquina)/(1 - CargaDaMaquina)$.

Quando uma tarefa é alocada a uma máquina, o valor do TBA correspondente a esta máquina é acrescentado de acordo com o $CustoDaTarefa$. As Tarefas são alocadas até todas as máquinas da grade estejam em uso. A seguir, inicia-se a execução das aplicações. Quando uma tarefa termina, todas as tarefas que não estão sendo executadas são desalocadas e re-escalonadas novamente. Este cenário continua até todas as tarefas serem completadas [14] [15] [35].

Segundo Falavinha et al. [35], o algoritmo *Dynamic* FPLTF analisa a carga de processamento de todos os aglomerados de máquinas (*clusters*) da grade no momento de escalonar uma tarefa, não permitindo que uma tarefa muito grande seja atribuída a um aglomerado que possua um poder de processamento muito ruim, tentando minimizar os efeitos da dinamicidade da grade [14] [15] [35]. Portanto, a máquina que em um primeiro momento é muito rápida e possui pouca carga recebe mais tarefas que uma máquina muito rápida, mas que esteja com bastante carga.

Esta heurística é vantajosa quando se tem informações acerca do ambiente necessárias para sua execução, mas é complicado implementá-la na prática [15]. As informações requeridas (tamanho da tarefa, carga da máquina e velocidade da máquina) são difíceis de se obter e frequentemente não estão disponíveis devido a restrições administrativas.

3.3.5 *Sufferage*

A ideia básica da heurística *Sufferage* é determinar quanto cada tarefa seria prejudicada ("sofreria") se não fosse escalonada no processador que a executaria de forma mais eficiente [14] [15]. Esta heurística tem por objetivo otimizar o tempo de conclusão das tarefas. *Sufferage* prioriza as tarefas de acordo com o valor que mede o "sofrimento" de cada uma. Este valor é definido como a diferença entre o menor e o segundo menor tempo de execução previsto para a tarefa, considerando todos os processadores da grade. A tarefa com maior valor de *sufferage* prevalece. Se outra tarefa foi previamente atribuída à máquina, o valor de *sufferage* da tarefa previamente atribuída e da nova tarefa são comparadas. A tarefa que permanecerá na máquina é a que possui o maior valor.

O valor *sufferage* de cada tarefa muda durante sua execução, pois a dinamicidade de carga é intrínseca a grade [1]. Para lidar com isto, toda vez que uma tarefa termina, todas as tarefas que ainda não começaram sua execução serão re-mapeadas, obedecendo o algoritmo ao ser invocado novamente. Assim como o *Dynamic* FPLTF, o *Sufferage* precisa de muita informação do ambiente de execução e das tarefas [1].

3.3.6 *XSufferage*

XSufferage [8] é um exemplo de heurística para escalonar aplicações que utilizam dados intensivamente [17] e se baseia nas informações sobre o desempenho dos recursos e da rede (ex: largura de banda) que os interliga [1]. Trata-se de um algoritmo estático e heurístico, e é uma extensão da heurística de escalonamento *Sufferage*. O *XSufferage* é uma heurística do tipo *batch* cujo objetivo funcional é centrado em aplicações paramétricas (*Bag of Tasks*) e *Parameter Sweep*.

A principal diferença entre *Sufferage* e *XSufferage* é o método usado para calcular o valor de "sofrimento" [1] [17] [47]. No *XSufferage*, o valor de "sofrimento" é calculado a partir do tempo de conclusão mínimo para cada aglomerado e sobre todas as máquinas em cada aglomerado. Assim, para cada tarefa é calculado o valor de "sofrimento" no aglomerado, esse valor é a diferença entre o menor e o segundo menor valor.

O algoritmo de escalonamento *XSufferage* usa informações sobre os aglomerados, os quais precisam estar disponíveis no momento em que o algoritmo vai alocar as tarefas [1]. O tipo de informações que devem ser conhecidas sobre o ambiente de execução para escalonar as tarefas são: disponibilidade de CPU; disponibilidade da rede; e tempo de execução das tarefas. Estas informações devem ser conhecidas antes do escalonamento e são utilizadas para decidir qual tarefa deve ser escalonada em qual processador.

Um ponto importante a ser observado é que o algoritmo considera somente os recursos livres no momento em que vai escalonar uma tarefa, pois caso contrário sempre o recurso mais rápido e com a melhor conexão de rede receberia todas as tarefas [35]. Assim como o *Dynamic* FPLTF, o *XSufferage* também possui dificuldade de obter as informações sobre a disponibilidade dos recursos. Seu problema está relacionado ao fato de considerar apenas nós livres no momento do escalonamento.

A Figura 3.4 apresenta o pseudocódigo do algoritmo *XSufferage* [17]. Para cada tarefa da aplicação, é identificado o aglomerado no qual a tarefa terá o menor tempo de execução (Linhas 9 a 14). Com o aglomerado identificado, o valor de "sofrimento" é calculado levando-se em consideração os recursos com o menor tempo de execução e o segundo menor (Linhas 16 e 17). Por fim, quando todas as tarefas tiverem o seu valor de "sofrimento" calculado, a tarefa com o maior valor de "sofrimento" será escalonada (Linhas 19 a 26). O processo de escalonamento para quando não há mais tarefas a serem escalonadas (Linha 2).

```
1: function schedule(Grid grid, Job job)
2:   while job.hasTaskNotScheduled() do
3:     maxSufferage = MIN
4:     nextTask = null
5:     nextResource = null
6:     for Task task : job.getTaskNotScheduled() do
7:       bestCompletionTime = MAX
8:       bestSite = null
9:       for Site site : grid.getSites() do
10:        if getCompletionTime(task, site) < bestCompletionTime then
11:          bestCompletionTime = getBestCompletionTime(task, site)
12:          bestSite = site
13:        end if
14:      end for
15:
16:      secondBestCompletionTime = getSecondBestCompletionTime(task, bestSite)
17:      sufferage = secondBestCompletionTime - bestCompletionTime
18:
19:      if sufferage > maxSufferage then
20:        maxSufferage = sufferage
21:        nextTask = task
22:        nextResource = getResourceWithBestCompletionTime(task, bestSite)
23:      end if
24:    end for
25:
26:    assign(nextTask, nextResource)
27:
28:  end while
29: end function
```

Figura 3.4: Heurística *XSufferage*

3.3.7 *Workqueue*

O *Workqueue* é um algoritmo *on-line* que atribui cada tarefa para a primeira máquina que se tornar disponível. Se múltiplas maquinas se tornarem disponíveis simultaneamente, o algoritmo escolhe uma aleatoriamente, onde ele mapeia somente uma tarefa para um recurso. Trata-se de um algoritmo estático e heurístico utilizado para escalonar aplicações paramétricas (BoT).

O *Workqueue* não requer qualquer tipo de informação para escalonar tarefas [1], evitando o problema de obtenção de informações sobre a aplicação e os recursos da grade visto nos algoritmos anteriores.

Este algoritmo é particularmente útil para sistemas cujo objetivo principal é maximizar a utilização dos recursos, ao invés de minimizar o tempo de execução de tarefas individuais. Dependendo da implementação, este algoritmo pode necessitar

consultar o estado de todas as m máquinas para encontrar aquelas disponíveis. Sendo assim, a ordem de complexidade deste algoritmo para realizar um escalonamento é $O(m)$.

A ideia por detrás do *Workqueue* é que mais tarefas serão atribuídas para máquinas rápidas ou ociosas, enquanto que máquinas lentas ou ocupadas processarão uma pequena carga [1]. O problema surge quando uma tarefa grande é atribuída a uma máquina lenta no processo de escalonamento. Quando isso ocorre, a conclusão da aplicação será adiada até a execução completa desta tarefa.

3.3.8 *Workqueue* com Replicação

O *Workqueue* com Replicação (WQR) [15] é um algoritmo estático e heurístico do tipo *on-line* cujo objetivo funcional é centrado na execução de aplicações paramétricas (BoT) que utilizam dados intensivamente (I/O bound). Ele foi desenvolvido para solucionar o problema da obtenção de informações sobre a aplicação e a carga de utilização dos recursos da grade para realizar o escalonamento [47]. Este algoritmo de escalonamento é similar ao *Workqueue* na sua fase inicial, onde tarefas são escalonadas de forma aleatória em recursos ociosos na grade. A diferença entre essas duas heurísticas está no fato de que a heurística WQR realiza um processo de replicação de tarefas quando houver disponibilidade de recursos na grade. Este algoritmo é utilizado pelo *middleware OurGrid* [11].

A ideia básica do WQR é submeter réplicas das tarefas em execução aumentando a porcentagem de conclusão das tarefas enquanto houver recursos disponíveis [47]. O processo de replicação é feito com o objetivo de diminuir o *makespan*[1] da aplicação, evitando que uma tarefa fique executando por muito tempo em um recurso com pouco poder computacional [17]. A heurística WQR define um fator máximo para o número de réplicas de uma tarefa, visando diminuir o desperdício de recursos.

A Figura 3.5 apresenta o pseudocódigo da heurística WQR [17]. Esta heurística escalona as tarefas nos recursos de maneira aleatória (Linhas 2 a 7). Se não houver recurso disponível, o algoritmo espera até que um recurso fique disponível

[1] O *makespan* é a diferença entre o instante de tempo em que a primeira tarefa da aplicação inicia sua execução e o instante de tempo no qual a última tarefa da mesma termina sua execução.

(Linhas 3 a 5). Quando todas as tarefas da aplicação tiverem sido escalonadas e caso ainda haja recursos disponíveis, a heurística entra na fase de replicação (Linhas 9 a 17). Quando uma tarefa finaliza sua execução, todas as réplicas dessa tarefa são canceladas (Linhas 20 a 22).

```
1:  function schedule(Grid grid, Job job, int replicationFactor)
2:    while job.hasTaskNotScheduled() do
3:      if !grid.hasAvailableResource() then
4:        sleepUntilThereIsAvailableResource()
5:      end if
6:      assign(job.getNextUnscheduledTask, grid.getNextAvailableResource())
7:    end while
8:
9:    while job.hasAllTasksScheduled() AND job.hasRunningTask() do
10:     if !grid.hasAvailableResource() then
11:       sleepUntilThereIsAvailableResource()
12:     end if
13:     nextTask = job.getNextRunningTaskWithMinReplicas()
14:     if nextTask.getNumberOfReplicas() < replicationFactor then
15:       replicate(nextTask, grid.getNextAvailableResource())
16:     end if
17:   end while
18: end function
19:
20: function whenTaskFinishes(Task task)
21:   task.cancelAllRunningReplicas()
22: end function
```

Figura 3.5: Heurística *Workqueue* com Replicação

3.3.9 *Storage Affinity*

O algoritmo *Storage Affinity* [47] foi feito com o intuito de explorar a reutilização de dados para melhorar o desempenho de aplicações que utilizam grandes quantidades de informação [1]. Trata-se de um algoritmo estático e heurístico do tipo *batch*, baseada em replicação, cujo objetivo funcional é centrado na execução de aplicações paramétricas (*Bag of Tasks*) que utilizam dados de forma intensiva. Argumenta-se que aplicações BoT geralmente processam uma grande quantidade de dados, reutilizando uma porção significativa dos dados de entrada [17]. Como exemplo destas aplicações, temos aquelas relacionadas a visualização de resultados de experimentos científicos.

O modelo da grade computacional pressuposto por esta heurística é baseado em um ou mais aglomerados. Cada aglomerado é composto por recursos computacionais que executam tarefas e um recurso para armazenamento de dados.

Todos os recursos computacionais devem ter acesso ao recurso para armazenamento de dados do seu aglomerado [17].

O método de escalonamento de tarefas é definido sobre o conceito constante do nome dado ao algoritmo, a afinidade. O valor da afinidade entre uma tarefa e um aglomerado determina quão próximo do aglomerado esta tarefa está. A semântica do termo "próximo" está associada a quantidade de bytes da entrada da tarefa que já está armazenada remotamente em um dado aglomerado. Assim, quanto mais bytes de entrada da tarefa estiver armazenado no aglomerado, mais próximo a tarefa estará do mesmo, pois possui mais afinidade de armazenamento (*storage affinity*) [1] [35].

Além de aproveitar a reutilização de dados, o *Storage Affinity* também realiza replicação de tarefas, com a esperança que as replicas tenham a chance de serem submetidas a processadores mais rápidos do que aqueles associados à tarefas originais, reduzindo o tempo de execução da aplicação [1] [35].

A heurística calcula a afinidade de cada tarefa com todos os recursos, mapeando a tarefa com a maior afinidade ao recurso que este valor foi obtido [17]. Este processo é repetido de maneira iterativa até que todas as tarefas sejam escalonadas. Como na heurística WQR, o *Storage Affinity* também aplica a técnica de replicação. Entretanto, o *Storage Affinity* segue um critério específico para criar uma réplica, diferentemente de WQR que replica as tarefas de maneira aleatória. O *Storage Affinity* somente replica uma tarefa se algum outro recurso do mesmo aglomerado estiver ocioso, evitando que os dados de entrada da tarefa sejam novamente transferidos, pois eles já estão lá armazenados [17].

A Figura 3.6 apresenta o pseudocódigo da heurística *Storage Affinity* [17]. Para cada tarefa da aplicação, é selecionado um aglomerado cuja tarefa tem o maior valor de afinidade. A cada iteração do laço, a tarefa que possuir o maior valor de afinidade é escalonada (Linhas 2 a 17). Quando não existir mais tarefas para serem escalonadas, a heurística entra na fase de replicação (Linhas 19 a 28).

```
1: function schedule(Grid grid, Job job, int replicationFactor)
2:     while job.hasTaskNotScheduled() do
3:         for Task task : job.getTaskNotScheduled() do
4:             if !grid.hasAvailableResource() then
5:                 sleepUntilThereIsAvailableResource()
6:             end if
7:             for Site site : grid.getSitesWithAvailableResource() do
8:                 tempAffinity = calculateAffinity(task, site)
9:                 if tempAffinity > maxAffinity then
10:                    maxAffinity = tempAffinity
11:                    nextTask = task
12:                    nextSite = site
13:                end if
14:            end for
15:        end for
16:        assign(nextTask, nextSite.getAvailableResource())
17:    end while
18:
19:    while job.hasTaskNotFinished() do
20:        if !grid.hasAvailableResource() then
21:            sleepUntilThereIsAvailableResource()
22:        end if
23:        nextTask = job.getNextRunningTaskWithMinReplicas())
24:        if nextTask.getNumberOfReplicas() < replicationFactor then
25:            Site site = grid.getBestAffinitySitesWithAvailableResource(nextTask).nextSite()
26:            replicate(nextTask, site.getAvailableResource())
27:        end if
28:    end while
29: end function
30:
31: function whenTaskFinishes(Task task)
32:    task.cancelAllRunningReplicas()
33: end function
```

Figura 3.6: Heurística *Storage Affinity*

3.3.10 RUMR

RUMR [17] é um algoritmo estático e heurístico do tipo *batch*, cujo objetivo funcional é centrado no escalonamento de aplicações paramétricas (BoT) que utilizam dados de forma intensiva, especificamente aquelas que possuem carga de trabalho divisível (*divisible workloads*). Normalmente, em aplicações com a carga divisível, a heurística de escalonamento fica responsável por dividir os dados de entrada em pedaços menores formando as tarefas da aplicação.

Na heurística RUMR, tarefas com granularidades pequenas (tamanho pequeno de dados de entrada) são inicialmente formadas e escalonadas, com o objetivo de colocar os recursos para processar o mais rápido possível, evitando que recursos

fiquem por muito tempo ociosos aguardando a transferência de dados. Durante o escalonamento, RUMR aumenta a granularidade das tarefas enquanto a aplicação está executando. Ao final da execução da aplicação, a granularidade da tarefa diminui visando evitar o escalonamento de tarefas longas no final da execução. Em uma grade computacional, escalonar tarefas grandes no final da execução não é uma boa ideia, pois uma tarefa grande pode ser associada a um recurso sobrecarregado, desta forma afetando substancialmente no tempo de execução da aplicação.

A Figura 3.7 apresenta o pseudocódigo da heurística RUMR [17]. No início, o algoritmo cria um conjunto de tarefas iniciais com granularidade pequena (Linha 2) a partir da aplicação submetida para execução. As tarefas são, então, escalonadas para os recursos (Linhas 9 a 13). Depois, a heurística entra na fase UMR, calculando-se inicialmente o número de iterações a serem realizadas (Linha 16). Nesta fase, a cada iteração a granularidade das tarefas que os recursos recebem aumenta (Linhas 17 a 21). Quando a última iteração da fase UMR termina, a heurística entra em uma fase denominada *Weighted Factory*. Nesta fase, a granularidade das tarefas vai diminuindo a cada iteração até que toda a carga da aplicação seja processada (Linhas 28 a 38).

```
1: function schedule(Grid grid, Job job)
2:     job.createInitialTasks()
3:
4:     scheduleFirstRound(grid, job)
5:     scheduleUMRPhase(grid, job)
6:     scheduleWFPhase(grid, job)
7: end function
8:
9: function scheduleFirstRound(Grid grid, Job job)
10:    for Task task : job.getTasksNotScheduled() do
11:        assign(task, task.getResource())
12:    end for
13: end function
14:
15: function scheduleUMRPhase(Grid grid, Job job)
16:    umrIteractions = getNumberOfUMRIteractions(grid, job)
17:    for int i = 1; i <= umrIterations; i++ do
18:        for Resource resource : grid.getResources() do
19:            job.addTask(createNextUMRPhaseTask(resource))
20:        end for
21:    end for
22:
23:    for Task task : job.getTasksNotScheduled() do
24:        assign(task, task.getResource())
25:    end for
26: end function
27:
28: function schedulerWFPhase(Grid grid, Job job)
29:    while !job.getWorkload().isCompleteProcessed() do
30:        for Resource resource : grid.getResources() do
31:            job.addTask(createNextWFPhaseTask(resource))
32:        end for
33:    end while
34:
35:    for Task task : job.getTasksNotScheduled() do
36:        assign(task, task.getResource())
37:    end for
38: end function
```

Figura 3.7: Heurística RUMR

3.4 Escalonamento de Aplicações de Acordo com Requisitos de QoS Especificados

Diferentemente dos algoritmos vistos na Seção 3.3, veremos, nesta Seção, trabalhos que descrevem algoritmos que tem por propósito atender a requisitos de Qualidade de Serviço (QoS) especificados pelo usuário da grade de computadores [6] [54] [36]. Estes trabalhos apresentam novas abordagens de escalonamento, ou a adequação das abordagens já existentes, para que as mesmas passem a considerar os requisitos de QoS como métricas para a avaliação de estratégias de escalonamento em grades de computadores.

Buyya et al. [6], apresentam um algoritmo de escalonamento para aplicações do tipo *Parameter Sweep* em grades globais[2]. O objetivo do algoritmo apresentado é escalonar aplicações para recursos considerando, quando possível, a melhor relação custo benefício entre o tempo para a execução das aplicações e o custo financeiro para realizar estas computações. Desta forma, se várias máquinas oferecem o mesmo tempo de conclusão para uma aplicação, essa deverá ser escalonada para aquela que apresentar o menor custo (Requisito de QoS) para utilização dos recursos. O algoritmo é denominado *Deadline and Budget Constrained* (DBC) *cost-time optimisation*.

O usuário de uma grade que utilize o algoritmo de escalonamento proposto fornece, opcionalmente, os seguintes parâmetros ao solicitar a execução de uma aplicação: o tempo que está disposto a esperar até a conclusão da aplicação (*deadline*) e o preço que está disposto a pagar pela realização desta computação (*budget*). O algoritmo é do tipo *on-line*, e executará enquanto houverem aplicações a serem escalonadas e que ainda tenham condições de serem executadas de acordo com o *deadline* e o custo definidos por seus usuários. O algoritmo foi avaliado utilizando-se o simulador *GridSim*.

[2]Na computação em grades globais, usuários e provedores de recursos organizam várias OVs para compartilhar recursos e serviços.

Yu e Buyya [54], apresentam um algoritmo de escalonamento genético[3] para aplicações de *workflow*. O objetivo desse algoritmo é escalonar as aplicações dentro de um determinado prazo com o menor custo financeiro possível (requisitos de QoS).

O artigo abre discussões sobre as diferenças entre grades utilitárias e grades comunitárias. As grades utilitárias são aquelas nas quais os usuários precisam pagar para garantir a reserva dos recursos e a qualidade do serviço, sendo que o algoritmo proposto em [54] se aplica a esse tipo de grade. Em grades utilitárias, por exemplo, o preço da utilização dos recursos pode ser determinado pela velocidade de processamento. Já a grade comunitária é aquela na qual o acesso é livre, no entanto a disponibilidade dos serviços não pode ser garantida.

Em uma visão geral do problema, o autor trabalha com aplicações *Workflow* modeladas segundo um grafo acíclico direcionado (*Directed Acyclic Graph* (DAG)). Uma característica desse tipo de aplicação é que uma tarefa filha não pode executar até que a tarefa pai tenha completado a sua execução, existindo assim uma dependência hierárquica entre as tarefas. Cada tarefa que compõe a aplicação de *Workflow* pode estar associada a somente um tipo de serviço disponibilizado pela grade utilitária.

Basicamente são providos dois tipos de serviços pela grade utilitária: *Resources Services* e *Applications Services*. O primeiro está relacionado a oferta de recursos de hardware como serviços, tais como memoria, processador, armazenamento, etc. O segundo está relacionado a oferta de aplicações especializadas.

A fim de avaliar a abordagem proposta, os autores implementaram o algoritmo e compararam-no com um conjunto de heurísticas não-genéticas para dois tipos diferentes de aplicações de *Workflow*, balanceadas e não-balanceadas, em um ambiente de grade simulado, utilizando o simulador *GridSim*.

Kyong Kim et al. [36] abordam uma problemática que tem se tornado cada vez mais relevante em ambientes de computação em aglomerados: a necessidade de se alcançar uma maior economia de energia para a execução de aplicações, o que deu origem ao termo *Power-aware scheduling*.

[3] Algoritmos genéticos (AGs) fornecem as técnicas de pesquisa robustas que permitem encontrar uma solução ótima dentro de um grande espaço de busca em tempo polinomial, aplicando o princípio da evolução. Um algoritmo genético combina a exploração das melhores soluções a partir de pesquisas anteriores com a exploração de novas regiões do espaço de solução. Qualquer solução no espaço de busca do problema é representada por um indivíduo (cromossomos). O algoritmo genético mantém uma população de indivíduos que se desenvolve ao longo de gerações.

Existem duas razões principais para a necessidade de computação ciente do consumo de energia em sistemas baseados em aglomerados: o custo operacional e a confiabilidade do sistema. Um fator dominante no custo operacional dos *data centers* é o custo da energia consumida pelos servidores. Outro fator é o aumento da temperatura causada pelo alto consumo de energia, comprometendo a confiabilidade do sistema. Em altas temperaturas a computação estaria mais vulnerável dado o aumento da probabilidade de ocorrência de falhas de hardware e, consequentemente, comprometendo a execução de aplicações.

Os autores apresentam dois algoritmos de escalonamento para aplicações *Bag of Tasks*, um voltado para a política de alocação de recurso *Space-Shared* e outro para a política *Time-Shared*. A política *Space-Shared* permite a execução de apenas uma tarefa por vez em uma determinada unidade de processamento, enquanto que na política *Time-Shared* múltiplas tarefas compartilham a unidade de processamento, de forma que cada tarefa executa alternadamente em uma determinada fatia de tempo (*time-slice*).

O objetivo dos algoritmos é conseguir cumprir as restrições de tempo para a execução das aplicações minimizando o consumo de energia em sistemas baseados em aglomerados. Um pressuposto adotado é que os nós da grade possuam suporte a *Dynamic Voltage Scaling* (DVS), uma técnica de gerenciamento de energia que permite ajustar dinamicamente a tensão utilizada por um dado componente. Segundo os autores, os algoritmos propostos conseguem reduzir o consumo e energia a partir do controle que efetuam sobre os níveis de tensão elétrica dos nós da grade.

O que diferencia o trabalho proposto nesta dissertação de mestrado das abordagens anteriormente descritas é a atenção específica de nossa proposta a ambientes de grades oportunistas. Desta forma, foi levado em consideração a possível existência de carga de trabalho local nos nós que compõem a grade, sobre a qual não temos controle. Isto impede a previsão precisa do tempo de execução de tarefas nos nós da grade, forçando a existência de um mecanismo que acompanhe o progresso da execução das tarefas e a eventual necessidade de reescalonamento e migração das mesmas. Além disso, levamos em consideração que grades de computadores pessoais são sujeitos a diversos tipos de falhas e podem exibir problemas físicos (tanto de hardware quanto nos enlaces da rede), erros lógicos (na aplicação ou nos protocolos de comunicação, por exemplo) e sofrer invasões de software malicioso. Outro tipo usual de falha está relacionado ao dinamismo dos recursos, que usualmente não são

dedicados e podem subitamente tornarem-se indisponíveis, mesmo quando estiverem executando computações em favor da grade. Além disso, as aplicações usualmente são compostas por tarefas de longa duração, que podem levar horas ou até mesmo dias para serem executadas, o que aumenta significativamente a possibilidade de falhas [48]. Desta forma, foi integrado a nossa solução um mecanismo autonômico de tolerância a falhas na execução de aplicações desenvolvido para ambientes de grades oportunistas [52]. Este mecanismo é baseado no uso de *checkpoints* e permite o ajuste autonômico do intervalo de tempo entre salvas sucessivas do estado de execução das tarefas contribuindo, assim, para o aumento da taxa de sucesso na execução das aplicações e, ao mesmo tempo, reduzindo o custo envolvido no processo de *checkpoint*.

Finalmente, argumentamos que o dinamismo e a instabilidade dos recursos que compõem as grades oportunistas tornam seu ambiente de execução inadequado para a execução de aplicações que possuam restrições severas no seu tempo de execução (*hard-deadline*) e, portanto, nossa abordagem é voltada a atender restrições brandas relacionadas ao tempo de execução das aplicações (*soft-deadlines*).

3.5 Conclusões

Este capítulo apresentou os fundamentos do processo de escalonamento de aplicações em grades de computadores utilizados neste trabalho. Foram descritas as principais características de uma arquitetura básica de escalonamento em grades, compreendendo a submissão de aplicações, seleção de recursos e execução da aplicação. Além disso, foi apresentado uma taxonomia para os algoritmos de escalonamento em grades de computadores.

Foram apresentados importantes algoritmos de escalonamento utilizados no contexto de grades e as estratégias de escalonamento dos principais projetos de grade. Diferenciaram-se os algoritmos pelo modo de escalonamento (*on-line* e *batch*) e pelo uso ou não das informações sobre o tempo estimado de execução das tarefas e sobre as características de *hardware* dos provedores de recursos.

Finalmente, foram apresentados trabalhos que levam em consideração parâmetros de QoS no processo de escalonamento das aplicações. Destacaram-se as principais diferenças entre estas abordagens e a proposta descrita neste trabalho, que foi desenvolvida e avaliada levando-se em consideração características típicas de ambientes de grades computacionais oportunistas.

4 Abordagem Proposta

Este capítulo apresenta o mecanismo para o gerenciamento da execução de aplicações em *desktop grids* oportunistas que possuam restrições de tempo de execução definidas pelos usuários no ato de sua submissão. Esse mecanismo foi desenvolvido para ser incorporado ao *middlware* InteGrade e foi avaliado através de simulações. Neste capítulo serão detalhados os componentes do mecanismo proposto, que possui como componente principal uma heurística de escalonamento do tipo *on-line* desenvolvida para escalonar as aplicações com base nas restrições impostas pelo usuário. Esse algoritmo considera que existem duas classes de aplicações: *soft-deadline*, que designa as aplicações que possuem restrições quanto ao tempo máximo em que devem ser executadas; e *nice*, que designa aplicações que não possuem restrições de tempo de execução.

4.1 Arquitetura da Abordagem Proposta

A abordagem proposta neste trabalho foi desenvolvida no contexto do projeto InteGrade[1] [16], um esforço multi-institucional para o desenvolvimento de um middleware para grades computacionais oportunistas. Neste middleware, o usuário ao submeter uma aplicação para execução informa restrições e preferências a serem adotadas em sua execução. Restrições definem requisitos mínimos para a seleção dos nós a serem utilizados, como plataformas específicas de hardware e software. Preferências definem uma ordem na escolha dos recursos, como a execução em máquinas que tenham preferencialmente mais de 4 GB de memória principal disponível. Neste trabalho, consideramos também as preferências do usuário com relação a restrições de tempo para a execução das aplicações. A partir da abordagem proposta neste trabalho, o usuário passa a poder especificar se a aplicação sendo submetida pertence a classe *nice* ou *soft-deadline*. Neste último caso, o usuário deve prover um prazo desejado para a execução da aplicação. Para aplicações da classe *soft-deadline*, nossa abordagem possui como meta a conclusão das aplicações dentro do prazo especificado, mesmo em face às dificuldades encontradas no contexto de

[1]Homepage: `http://www.integrade.org.br`

grades oportunistas. Já para as aplicações da classe *nice*, a meta é apenas a conclusão da aplicação com sucesso.

Neste trabalho propomos um mecanismo de gerenciamento da execução de aplicações em *Desktop Grids* oportunistas. Grades de computadores pessoais oportunistas são ambientes computacionais muito dinâmicos. Existem diversos fatores que contribuem para que eles tenham essa característica. Dentre estes fatores, podemos citar as variações na taxa de disponibilidade de recursos (dado que os mesmos podem se registrar e sair da grade de forma imprevisível, de acordo com a necessidade de seus proprietários), as variações na taxa de ocorrência de falhas de recursos (por estes serem não-dedicados e controlados por seus usuários), no volume de aplicações submetidas para execução, no grau de heterogeneidade das tarefas que compõem as aplicações submetidas, entre outros. Esses aspectos aumentam a complexidade do problema de gerenciamento de execução de aplicações para este tipo de ambiente computacional.

O mecanismo de gerenciamento da execução de aplicações para grades oportunistas proposto nesta dissertação é composto pelos seguintes componentes:

1. Um mecanismo de predição do tempo de execução das aplicações baseado na capacidade de processamento dos recursos da grade.

2. Um mecanismo de tolerância a falhas na execução de aplicações baseado em *checkpointing*, cujo objetivo é garantir a execução com sucesso das aplicações, mesmo na ocorrência de falhas.

3. Um mecanismo que monitora o progresso da execução das tarefas com restrição de tempo em cada nó da grade, verificando a necessidade ou não de reescalonálas para outros nós de forma a atender o prazo estipulado.

4. Um mecanismo que monitora o tempo médio entre falhas de cada recurso.

5. Uma heurística de escalonamento que mapeia as tarefas que compõem a aplicação para nós que potencialmente possam cumprir o prazo de execução da aplicação especificado pelo usuário.

Dentre estes componentes, dois foram baseados em trabalhos anteriores: o mecanismo de predição e o mecanismo adaptativo de tolerância a falhas.

4.1.1 Mecanismo de Predição

A atual implementação do mecanismo de gerenciamento de execução de aplicações proposto neste trabalho de mestrado foi realizada através de uma ferramenta de simulação, sendo descrita no Capítulo 5. No simulador, a estimativa do tempo de execução de uma aplicação em um dado recurso é trivialmente obtida, já que tanto o tamanho da aplicação em Milhares de Instruções (MI) quanto a capacidade de cada recurso expressa em Milhares de Instruções por Segundo (MIPS) são conhecidas. No entanto, em um ambiente real esta estimativa é bem mais complicada de ser obtida e depende de diversos fatores. Algumas abordagens de predição do tempo de execução de aplicações em ambientes distribuídos heterogêneos, como as grades de computadores, foram propostas, tais como [39], [50] e [30]. Nas avaliações destas abordagens, seus autores relatam diversos experimentos que demonstram diferentes percentuais de acerto nas predições realizadas. Desta forma, para tornarmos as simulações de avaliação de nossa abordagem mais realistas, introduzimos uma taxa de erro no mecanismo de estimativa do tempo de execução das aplicações de acordo com o mecanismo de predição adotado em nosso trabalho que é baseado em Yan Liu [39]. Neste trabalho são apresentadas duas abordagens de predição de tempo de execução de aplicações. Na primeira abordagem calcula-se a estimativa do tempo de execução da aplicação baseado no registro de execuções anteriores da mesma ou de aplicações semelhantes. A segunda abordagem, utilizada em nosso trabalho, é baseada no conhecimento do modelo de execução da aplicação, onde o código da aplicação é analisado, estimando-se o tempo de execução de cada tarefa de acordo com a capacidade dos recursos da grade.

Atualmente o mecanismo proposto nesta dissertação é voltado para execução de aplicações regulares. Porém, aplicações paralelas e, em especial, aplicações paramétricas (*Bag of Tasks*) são comuns em grades de computadores. Considerando uma futura extensão do trabalho proposto nesta dissertação de mestrado para contemplar esta classe de aplicações, foram investigados também modelos de predição voltados para aplicações paralelas. Sun e Wu [50] desenvolveram um modelo matemático para prever o desempenho de um ambiente distribuído não-dedicado, baseado em Gong et al. [30]. O modelo de predição leva em consideração que as estações de trabalho que compõem o ambiente distribuído não são dedicadas, o

que é exatamente o caso da computação em grade oportunista. Deste modo, as tarefas paralelas submetidas à grade competem para a execução com tarefas sequenciais locais submetidas pelos usuários das máquinas. O modelo também considera que o sistema é composto por máquinas heterogêneas e serviços são distribuídos de forma heterogênea, separando a influência da utilização da máquina, taxa de serviço das tarefas sequenciais e atribuição de tarefas para execução em paralelo no cálculo do tempo de conclusão das aplicações. Um pressuposto do modelo proposto é que a tarefa pode ser particionada livremente em pequenos pedaços e não são levados em consideração os efeitos de sincronização, comunicação, processo de migração, ou da granularidade do paralelismo.

4.1.2 Mecanismo Adaptativo de Tolerância a Falhas

O mecanismo adaptativo de tolerância a falhas adotado em nosso trabalho é baseado em Viana et al. [52, 53]. Nestes trabalhos, os autores apresentam uma estratégia autonômica de tolerância a falhas durante a execução das tarefas submetidas por usuários de grades oportunistas, levando em consideração o dinamismo típico desses ambientes. Esse mecanismo é baseado em dois níveis de adaptações: (a) reconfigurações paramétricas das estratégias de tolerância a falhas adotadas (*checkpointing* e replicação); e (b) adaptações estruturais no mecanismo, substituindo completamente uma técnica de tolerância a falhas por outra. Neste trabalho de mestrado foram descritos de forma sucinta como funcionam esses dois níveis.

Primeiro Nível: Adaptação Paramétrica Usando *Checkpointing*

No primeiro nível de adaptação ajusta-se os parâmetros da técnica de tolerância a falhas em uso pelo *middleware* da grade de forma a manter o equilíbrio do sistema. A técnica de *checkpointing* naturalmente produz uma sobrecarga sobre o tempo de execução das tarefas, uma vez que é necessário interromper o processo a cada vez que é salvo o estado do progresso da aplicação. A periodicidade estática nas tomadas dos *checkpoints* das tarefas torna esta abordagem não muito vantajosa quando levamos em consideração a volatilidade dos recursos em *desktop grids*. Um recurso é dito volátil quando, em seu histórico de funcionamento, ele apresenta um grande número de falhas ou, em se tratando de *desktop grids*, o proprietário do recurso não o

disponibiliza com muita frequência para executar as tarefas da grade. Já os recursos estáveis são aqueles que são menos suscetíveis a falhas e, portanto, estão na maior parte do tempo disponíveis ou executando as computações submetidas à grade.

Nas abordagens estáticas, o *checkpointing* pode ser configurado com intervalos curtos para evitar que as tarefas, ao serem recuperadas das falhas, executem o mínimo possível para retornar ao mesmo estado de quando falharam. Contudo, essa configuração irá executar muitas vezes o procedimento que interrompe e salva o estado da tarefa, prolongando o tempo de sua conclusão. Quando o ambiente se torna estável em relação à disponibilidade dos recursos, essa sobrecarga é desnecessária. Por outro lado, em ambientes voláteis, a definição de longos intervalos para a tomada do *checkpointing* pode levar a muita reexecução de código da aplicação e, dependendo do grau de volatilidade dos recursos do ambiente, o *checkpointing* das tarefas pode nem chegar a ser feito. Conhecendo o grau de volatilidade dos recursos da grade, pode-se estimar intervalos de *checkpointing* que se ajustam melhor a cada situação.

A fim de reduzir a sobrecarga desnecessária causada pelo *checkpointing*, o modelo utilizado em [52, 53] ajusta os intervalos entre os *checkpoints* de cada tarefa de acordo com a volatilidade do recurso em que está executando. Então, se os processos executam em recursos estáveis, esse ajuste é feito de forma que os intervalos entre *checkpoints* são alargados e, portanto, é reduzida a quantidade de vezes que são feitos. De outra forma, se os processos executam em recursos voláteis, os intervalos são reduzidos e, consequentemente, uma maior quantidade de *checkpoint* será feita, garantindo que sejam salvos estados mais próximos do estado de execução no momento da falha.

Primeiro Nível: Reconfiguração Paramétrica na Replicação

Para alcançar um bom desempenho na técnica de replicação é preciso definir a quantidade de réplicas ideal a ser utilizada. Determinar um número fixo para a quantidade de réplicas na técnica de replicação em grades computacionais é uma tarefa difícil quando se utiliza uma abordagem estática. A geração de uma grande quantidade de réplicas a fim de aumentar a probabilidade de sucesso das tarefas pode saturar a grade a medida que novas tarefas vão sendo submetidas pelos usuários. Em

contrapartida, a geração de uma pequena quantidade de réplicas diminui as chances de conclusão das aplicações em razão da ocorrência de falhas.

Visando superar essa dificuldade, o modelo proposto em [52, 53] utiliza uma abordagem de adaptação paramétrica para ajustar a quantidade de réplicas usada na estratégia de replicação à medida que os nós da grade vão sendo alocados para execução das computações dos usuários. A abordagem funciona com base no percentual de ocupação de recursos. Uma vez que a ocupação cresce proporcionalmente ao número de réplicas empregado, a quantidade de réplicas é alterada quando são atingidas diferentes faixas desse percentual.

Segundo Nível de Adaptação: Reconfiguração Estrutural

O segundo nível de adaptação, é a realização de uma reconfiguração estrutural, ou seja, substitui-se a técnica de tolerância a falhas em uso. Para isto, o modelo proposto por Viana et al. [52, 53] possui um segundo nível de adaptação que prevê a troca da abordagem de tolerância a falhas entre duas possibilidades: o uso de replicação e a adoção de *checkpointing*. O uso da técnica de *checkpointing* impõe um custo adicional ao tempo de execução da aplicação em decorrência das paradas para realizar a obtenção e persistência do estado de execução das tarefas. A técnica de replicação não impõe este custo, mas requer maior uso de recursos da grade, podendo atrasar a execução de novas tarefas submetidas à mesma, dependendo da disponibilidade de seus recursos. Nesse nível, considera-se a possibilidade de se alternar dinamicamente entre as técnicas de replicação e *checkpointing*, de modo que possa explorar as vantagens dessas técnicas de acordo com o ambiente que for mais favorável para utilização de cada uma delas. O fator utilizado para a tomada de decisão neste nível de adaptação é o percentual de ocupação dos recursos. A abordagem inicia utilizando replicação quando o percentual de ocupação dos recursos é igual a zero. À medida que os recursos vão sendo ocupados com a execução de tarefas e réplicas, esse percentual cresce e os ajustes paramétricos na técnica de replicação passam a não produzir mais os efeitos desejados, tornando-se necessário a substituição desta técnica pelo *checkpointing*. Quando esses recursos vão sendo liberados para novas execuções, ou mais recursos vão se registrando na grade, o percentual diminui, tornando o ambiente novamente favorável à utilização da técnica de replicação.

No contexto do desenvolvimento do mecanismo de gerenciamento da execução de aplicações proposto neste trabalho de mestrado, reutilizamos o primeiro nível de adaptação do mecanismo adaptativo de tolerância a falhas proposto por Viana et al. [52, 53], a adaptação paramétrica usando a técnica de *checkpointing*. Portanto, a recuperação de falhas de execução de aplicações na grade é realizada através da recuperação do último estado salvo pelo mecanismo de *checkpointing* e o intervalo entre *checkpoints* é dinamicamente ajustado de acordo com a estabilidade do recurso da grade.

Para a implementação do mecanismo adaptativo de tolerância a falhas no InteGrade seria necessário: monitorar o MTBF dos nós usando os dados fornecidos pelo *Local Resource Manager* (LRM) e enviados ao *Global Resource Manager* (GRM); definir um *callback* a partir do código instrumentado para tomada de *checkpoints* nas aplicações, que receberia como parâmetro o intervalo entre *checkpoints* a ser utilizado; alterar o LRM que já possui a atribuição de gerenciar a execução das aplicações em seu nó para que ele atualizasse através do *callback* o intervalo de checkpoint a ser utilizado pelas aplicações em execução.

4.1.3 Mecanismos de Monitoramento

A abordagem proposta neste trabalho pressupõe a monitoração do progresso da execução das tarefas nos recursos da grade e o tempo médio entre falhas de cada recurso, descritos a seguir.

Progresso da Execução de Tarefas

De acordo com a arquitetura básica do escalonamento de aplicações em grades de computadores, vista na Figura 3.1 da Seção 3.1.1, o Escalonador da Grade (EG) seleciona os recursos a partir das restrições e preferências providas pelos usuários. O Serviço de Informação da grade (SIG) provê um conjunto de informações para os escalonadores da grade. As informações de cada recurso são constantemente monitoradas pelo Gerenciador de Recursos Locais (GRL) e são enviadas periodicamente para o SIG. No caso do middleware InteGrade, o componente LRM realiza o papel do GRL de coletar informações sobre o recurso monitorado. Além disto, este componente é responsável por receber as aplicações enviadas pelo

escalonador da grade e preparar o ambiente para sua execução. Após a preparação do ambiente, o LRM inicia a execução das aplicações. Quando as aplicações terminam sua execução, ele envia os resultados da execução das aplicações para os clientes da grade.

Para a implementação do mecanismo que monitora o progresso da execução de tarefas no InteGrade seria necessário: propor uma extensão do papel do componente LRM, instanciado em cada nó que executa aplicações a favor da grade. Este componente recebe do EG requisições para a execução de tarefas com suas respectivas restrições e propriedades, incluindo sua classe (*nice* ou *soft-deadline*) e o prazo estipulado para sua conclusão. As tarefas em execução devem notificar periodicamente o LRM a respeito do progresso de sua execução utilizando uma *Application Programing Interface* (API) bem definida. A cada notificação recebida, o LRM estima se a conclusão da aplicação deve ocorrer dentro do prazo estipulado e, em caso negativo, notifica a aplicação através de um método de *callback* forçando sua interrupção e a salva de estado em um armazém estável. A requisição para execução da aplicação é então reenviada ao escalonador, que irá mapeá-la para um outro nó da grade capaz de atender o prazo solicitado, caso disponível. Caso contrário, a aplicação é recusada e o usuário é informado que não foi possível cumprir o deadline da aplicação.

Tempo Médio entre Falhas de Recursos

No caso do middleware InteGrade, o componente LRM periodicamente envia dados coletados no recurso monitorado (como uso de CPU e memória principal disponível) para o componente GRM, equivalente ao Serviço de Informação da Grade (SIG) da Figura 3.1. Portanto, a ausência de uma mensagem de atualização de informações por um dado intervalo de tempo leva o GRM a concluir que o recurso monitorado pelo LRM correspondente falhou e um procedimento de recuperação das aplicações que estavam em execução no mesmo é automaticamente disparado.

Para a implementação do mecanismo que monitora o tempo médio entre falhas dos recursos no InteGrade seria necessário: propor uma extensão do papel do GRM, que passará a manter o histórico de falhas dos recursos da grade, calculando e disponibilizando a informação do MTBF dos mesmos para a heurística de mapeamento proposta, apresentada a seguir.

4.1.4 Heurística de Mapeamento Proposta

A heurística de escalonamento proposta neste trabalho é dinâmica, realizando o escalonamento e reescalonamento das aplicações de acordo com a disponibilidade dos recursos e do poder de processamento disponível. A heurística proposta é também uma heurística do tipo *on-line*, podendo mapear mais de uma tarefa para execução em um dado nó da grade. O mapeamento de tarefas da classe *soft-deadline*, ilustrado na Figura 4.1, é realizado levando-se em consideração o intervalo médio entre falhas (MTBF) dos nós e a sua capacidade de processamento. Escalona-se uma tarefa da classe *soft-deadline* para nós identificados como estáveis (que apresentem maior MTBF) e cuja capacidade permita concluir a execução da tarefa dentro do prazo estipulado pelo usuário ao submeter a aplicação para execução na grade. Devido ao custo imposto pelo uso da abordagem de *checkpointing* e a eventual carga de trabalho local do nó, considera-se aptos para realizar a tarefa os nós cuja capacidade de processamento estimada pelo algoritmo de predição adotado permita a execução da tarefa acrescida de uma margem de 10%. Caso a carga local de trabalho exceda esta estimativa inicial, o mecanismo de acompanhamento do progresso da execução das tarefas é responsável por identificar a incapacidade do nó em cumprir o prazo estipulado e re-submeter a tarefa para um novo escalonamento.

A abordagem proposta integra um mecanismo de reserva antecipada de recursos que funciona da seguinte forma: ao se realizar o mapeamento de uma tarefa, caso não haja nós disponíveis que atendam aos critérios anteriormente descritos, busca-se nós que já estejam executando tarefas e que atendam aos seguintes critérios: (a) se a tarefa for da classe *nice*, ela será interrompida para dar lugar a tarefa *soft-deadline*, reservando-se o recurso para que posteriormente possa dar continuidade à execução da tarefa *nice* interrompida, o que será realizado a partir do seu último *checkpoint* salvo (linhas 4 a 7 e linhas 28 a 31); (b) caso a tarefa seja da classe *soft-deadline*, verifica-se se o tempo restante para sua execução acrescido do tempo necessário para executar a tarefa sendo escalonada é suficiente para atender ao prazo definido para a execução desta última. Caso seja, o recurso é reservado para a tarefa em questão (linhas 9 a 12 e linhas 33 a 36). Finalmente, caso não seja encontrado um recurso que atenda o prazo estipulado para a execução da tarefa conforme os critérios descritos, o usuário terá sua aplicação recusada (linhas 13 a 16 e linhas 37 a 40).

Figura 4.1: Mapeamento *SOFT*

```
1: case SOFT
2: do
3:    if freeLRMs.size() == 0 then
4:       lrmNice = searchNiceLRM(lrmList, task, deadline);
5:       if lrmNice != null then
6:          maps(task, lrmNice);
7:          return mappingResult;
8:       else
9:          lrmReserved = reserveSoftLRM(lrmList, task, deadline);
10:         if lrmReserved != null then
11:            maps(task, lrmReserved);
12:            return mappingResult;
13:         else
14:            submissionRefused(task);
15:            return mappingResult;
16:         end if
17:      end if
18:   else
19:      for i = 0; i < freeLRMs.size(); i++ do
20:         lrm = freeLRMs.get(i);
21:         isMapped = canAchieveDeadline(lrm, task, deadline);
22:         if isMapped then
23:            maps(task, lrm);
24:            freeLRMs.remove(lrm);
25:            return mappingResult;
26:         else
27:            if i == freeLRMs.size() - 1 then
28:               lrmNice = searchNiceLRM(lrmList, task, deadline);
29:               if lrmNice != null then
30:                  maps(task, lrmNice);
31:                  return mappingResult;
32:               else
33:                  lrmReserved = reserveSoftLRM(lrmList, task, deadline);
34:                  if lrmReserved != null then
35:                     maps(task, lrmReserved);
36:                     return mappingResult;
37:                  else
38:                     submissionRefused(task);
39:                     return mappingResult;
40:                  end if
41:               end if
42:            end if
43:         end if
44:      end for
45:   end if
46: end case
```

A Figura 4.2 ilustra o algoritmo utilizado para o mapeamento de tarefas da classe *nice*. O mapeamento destas tarefas também é realizado levando-se em consideração o MTBF dos nós. No entanto, ao contrário do que é feito com tarefas da classe *soft-deadline*, escalona-se tarefas da classe *nice* para nós identificados como instáveis (que apresentem menor MTBF). Para tanto, gera-se a lista de nós disponíveis

ordenando-a pelo MTBF em ordem decrescente, mapeando a tarefa *nice* para o último recurso da lista (linhas 3 a 6). Caso não haja nós disponíveis, busca-se aqueles que já estejam executando tarefas da classe *nice*, escolhendo o primeiro para a execução da tarefa, realizando-se uma reserva antecipada deste recurso (linhas 7 a 11). Finalmente, caso não seja encontrado um recurso disponível ou que esteja executando uma tarefa da classe *nice*, o algoritmo irá mapear a tarefa para um recurso aleatório (linhas 13 a 16), colocando-a no final de sua fila de execução.

Figura 4.2: Mapeamento *NICE*

```
1: case NICE
2: do
3:    if freeLRMs.size() > 0 then
4:       lrm = freeLRMs.removeLast();
5:       maps(task, lrm);
6:       return mappingResult;
7:    else
8:       lrmReserved = reserveNiceLRM(resList);
9:       if lrmReserved != null then
10:         maps(task, lrmReserved);
11:         return mappingResult;
12:      else
13:         indexLRM = random.nextInt(lrmList.size());
14:         lrm = lrmList.get(indexLRM);
15:         maps(task, lrm);
16:         return mappingResult;
17:      end if
18:   end if
19: end case
```

4.2 Conclusões

Este capítulo apresentou a abordagem proposta neste trabalho de mestrado para o gerenciamento de aplicações com restrições de tempo de execução desenvolvido para ambientes de grades oportunistas e sua arquitetura. A abordagem proposta considera duas classes de aplicações: (a) *soft-deadline* e (b) *nice*. As aplicações da classe *soft-deadline*, possuem como meta a conclusão das aplicações dentro deste prazo, já para as aplicações da classe *nice* a meta é apenas a conclusão da aplicação com sucesso.

Neste trabalho, foi visto que a arquitetura do mecanismo de gerenciamento de aplicações é composta por cinco mecanismos: um mecanismo de predição do tempo de execução das aplicações, uma heurística de escalonamento, um mecanismo que monitora o progresso da execução das tarefas com restrição de tempo em cada nó da grade, um mecanismo que monitora o tempo médio entre falhas de cada recurso e um mecanismo de tolerância a falhas na execução de aplicações baseado em *checkpointing*. Dentre estes mecanismo temos como componente principal a heurística de escalonamento implementada no simulador AGST [18, 19].

5 Avaliação da Abordagem Proposta

Este capítulo descreve a avaliação realizada da abordagem proposta nesta dissertação de mestrado. Esta avaliação foi realizada utilizando-se uma ferramenta de simulação especificamente desenvolvida para ambientes de grades de computadores, o *Autonomic Grid Simulator Tool* (AGST). O uso da abordagem de simulação deve-se às inúmeras dificuldades para se obter um ambiente adequado para realizar experimentos que tenham por objetivo testar e avaliar a abordagem desenvolvida, tais como: a dificuldade em se ter acesso a um ambiente de grade com recursos em grande escala; a dificuldade em se explorar cenários com aplicações típicas de computação em grade, que podem executar durante várias horas e até mesmo dias, tornando inviável executar diversos testes considerando restrições de tempo; a dificuldade de se explorar cenários com recursos e aplicações envolvendo diversos usuários de forma repetitiva e controlada, devido à natureza dinâmica dos ambientes de grade e a dificuldade de coordenar os usuários. Neste capítulo foram apresentados uma breve descrição da ferramenta de simulação utilizada, o desenvolvimento do modelo de simulação que implementa a abordagem proposta e foram discutidos os resultados obtidos a partir das diversas simulações realizadas, que levaram em consideração diversos cenários típicos de ambientes de grades computacionais oportunistas.

5.1 A Ferramenta AGST

O AGST[1] [18, 19] é um simulador de eventos discretos orientado a objetos cujo principal objetivo é auxiliar desenvolvedores de *middleware* para grades oportunistas na validação de novos conceitos e suas implementações. Ele foi projetado para levar em consideração a dinâmica de grades oportunistas, provendo um conjunto de funcionalidades que agilizam o desenvolvimento de simulações que levam em consideração o dinamismo do ambiente de execução. Apesar de ter sido desenvolvido no contexto do projeto InteGrade, o AGST foi projetado para permitir a simulação de grades oportunistas de uma maneira geral, podendo ser aplicado a outros projetos de pesquisa envolvendo *middleware* de grades.

[1] `http://www.lsd.ufma.br/~agst`

A arquitetura do AGST é dividida em camadas, conforme ilustrado na Figura 5.1. Esta abordagem facilita a integração de novos componentes ou camadas.

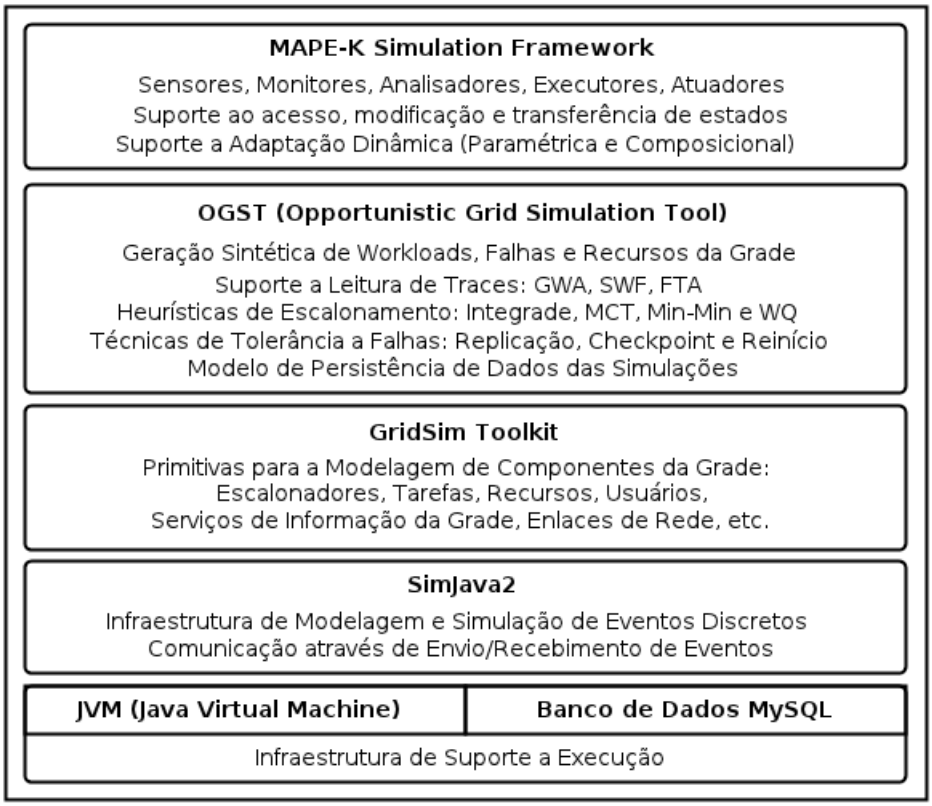

Figura 5.1: Arquitetura do AGST (Autonomic Grid Simulation Tool)

A primeira camada corresponde a infraestrutura de suporte a execução do AGST. Essa camada é composta pela *Java Virtual Machine* (JVM) e pelo Sistema Gerenciador de Banco de Dados (SGBD) MySQL. A JVM é necessária dado que o código das camadas superiores é escrito na linguagem Java. Já o SGBD MySQL é utilizado por padrão para armazenar os dados obtidos durante a execução das simulações.

A segunda camada corresponde ao pacote de simulação de propósito geral baseado em eventos discretos responsável pela interação entre as entidades da simulação, o SimJava2. Essa camada permite que os componentes das camadas superiores se comuniquem uns com os outros através de eventos (`Sim_Event`).

A terceira camada corresponde ao GridSim, que fornece para as camadas superiores os componentes primitivos necessárias para criação e gerenciamento de componentes do ambiente de grade, tais como: tarefas (`Gridlet`), recursos computacionais (`GridResource`), enlaces de rede (`Link`), usuários (`GridUser`), e o

serviço de informações da grade (`AbstractGIS`). Qualquer outro componente que precise se comunicar com outros componentes através do envio e recebimento de eventos, tal como um escalonador, pode ser implementado pelas camadas inferiores através de extensões da classe `GridSim`.

A quarta camada corresponde ao *Opportunistic Grid Simulator Tool* (OGST), um simulador de grades oportunistas. Essa camada faz uso dos componentes primitivos fornecidos pela camada inferior (GridSim) e a partir deles implementa componentes específicos do modelo de grades do OGST, tais como: Escalonador (`GridScheduler`), o Serviço de Informações da Grade `ResourceDataStorage`, e a Ferramenta de Submissão de Aplicações (`UserApplicationSubmitionTool`). Essa camada provê ainda mecanismos flexíveis para a geração automatizada de aplicações, recursos computacionais e enlaces de rede, fornecendo também uma biblioteca de heurísticas de escalonamento de tarefas comumente utilizadas por *middlewares* de grades reais: MCT, Min-min, Work-Queue, e InteGrade. Outras heurísticas podem ser facilmente implementadas através de extensões ao componente *Schedule Strategy*, provido pelo OGST. Também é fornecido por essa camada um conjunto de estratégias para a tolerância a falhas na execução de aplicações: *checkpointing*, reinício e replicação. Esta camada é ainda responsável pela geração automática do esquema utilizado pelo banco de dados (primeira camada), provida pelo componente `SimulationRecordDatabaseManagement`. Mais informações sobre o OGST podem ser encontradas nos trabalhos [13, 22, 29].

A quinta, e última camada, corresponde ao MAPE-K Simulation Framework. Essa camada fornece um conjunto básico de componentes extensíveis e reutilizáveis para modelagem e simulação de ciclos de gerenciamento autonômico em grades, e provê ao AGST os seguintes mecanismos: I) gerenciamento das operações de acesso e modificação das variáveis de estados dos elementos gerenciados; II) execução de ações de reconfiguração dinâmicas no ambiente de grade e; III) gerenciamento da comunicação e da interação entre componentes do ciclo autonômico. Mais informações sobre esta camada podem ser encontradas em [18, 19].

Descrevemos a seguir as principais funcionalidades providas pelo AGST utilizadas para o desenvolvimento deste trabalho.

5.1.1 Geração Automatizada de Aplicações e Tarefas

O GridSim não explicita um modelo de aplicações, fornecendo apenas a entidade básica para representar uma tarefa denominada *Gridlet*. Portanto, a geração de aplicações no GridSim não é automática. Extensões providas pelo AGST ao GridSim permitem gerar automaticamente dois tipos de aplicações: regulares (constituídas de uma única tarefa) e *Bag-of-Tasks* (constituídas de várias tarefas que executam paralelamente de forma independente umas das outras, ou seja, sem trocar dados durante o processo de execução). O tamanho das tarefas é definido em Milhões de Instruções (MIs).

No AGST, as aplicações podem ser geradas de três formas: (1) sinteticamente, através de distribuições de probabilidade (uniforme, *poisson*, exponencial e hiperexponencial) utilizadas na geração do tamanho das tarefas e de seus respectivos tempos de chegada no processo de simulação; (2) a partir de *traces* de *workloads* nos formatos *Grid Workload Format* (GWF)[2] e *Standart Workload Format* (SWF)[3]; e (3) através do modelo probabilístico de Lublin [40].

5.1.2 Geração Automatizada de Recursos Computacionais e Enlaces de Rede

O GridSim fornece as entidades necessárias para a criação de recursos da grade, tais como máquinas e processadores. Extensões ao GridSim permitem gerar esse recursos de forma automatizada. Para isso, alguns parâmetros devem ser informados, tais como: a quantidade de nós a serem gerados, a distribuição de probabilidade que melhor modela a heterogeneidade desejada dos recursos e o poder de processamento (definido em Milhões de Instruções Por Segundo (MIPS)). O percentual de uso do processador disponível para a execução de aplicações da grade pode ser definido pelo usuário gerado a partir de *traces* de *hostload*. AGST permite a interconexão de diversos aglomerados de recursos (*sites*) através da rede.

[2]http://gwa.ewi.tudelft.nl/pmwiki/
[3]http://www.cs.huji.ac.il/labs/parallel/workload/swf.html

5.1.3 Falhas de Recursos e Técnicas de Tolerância a Falhas

Falhas de recursos podem ser geradas sinteticamente ou a partir de bancos de dados de *traces* de falhas coletados de ambientes reais no padrão *Failure Trace Archive* (FTA)[4] [37]. A geração sintética de falhas é baseada em distribuições de probabilidade, tais como exponencial, hiperexponencial e weibull [10]. Deve-se fornecer os parâmetros de acordo com a distribuição utilizada. Por exemplo, no caso de uma distribuição exponencial, devem ser fornecidos o tempo médio entre falhas (MTBF) e o tempo para recuperação (*Mean Time To Recovery* (MTTR)) dos recursos.

No AGST, a recuperação da execução de aplicações em decorrência de falhas de nós pode ser baseada na técnica do reinício (tarefas em execução em um nó que falhe são reexecutadas do início em outros nós da grade) ou no uso de pontos de controle (*checkpointing*). É possível ainda simular a execução de aplicações com replicação, técnica comumente usada em grades.

5.1.4 Simulação de Comportamento Autonômico

As grades de computadores são sistemas que possuem grande escalabilidade, alta heterogeneidade e ambiente de execução dinâmico. Essas características tornam o gerenciamento desses sistemas altamente complexo. A computação autonômica agrega vários campos da computação com o objetivo de dotar sistemas complexos com a capacidade de auto-gerenciamento.

A essência do sistema autonômico é o auto-gerenciamento. Para implementá-lo, o sistema deve ao mesmo tempo estar atento a si próprio e ao seu ambiente. Desta forma, o sistema deve conhecer com precisão a sua própria situação e ter consciência do ambiente operacional em que atua. Do ponto de vista prático, para realizar a visão original do termo computação autonômica [42] os sistemas autonômicos precisam possuir as seguintes propriedades: autoconfiguração, auto-otimização, autocura e autoproteção.

Em 2003, a *International Business Machines* (IBM) propôs uma versão automatizada do ciclo de gerenciamento de sistemas chamado de *Monitoring, Analysis, Planning, Execution and Knowledge* (MAPE-K) [33]. Este modelo está sendo cada

[4] http://fta.inria.fr/

vez mais utilizado para inter-relacionar os componentes arquiteturais dos sistemas autonômicos.

O simulador de grades autonômicas AGST, é implementado com base na arquitetura MAPE-K, e, portanto, oferece um arcabouço que implementa essa arquitetura, provendo suporte a todas as fases de um ciclo de gerenciamento autonômico MAPE-K: monitoramento de recursos do ambiente de grade por meio de sensores; análise das informações de contexto; controle e execução de ações de reconfiguração dinâmica através de atuadores. O AGST oferece suporte a execução de dois tipos de adaptação dinâmica: paramétrica e composicional. A adaptação paramétrica consiste na alteração de variáveis que determinam o comportamento de algoritmos utilizados pelo *middleware* de grade. Já a adaptação composicional consiste na troca de algoritmos ou partes estruturais do *middleware*, permitindo que este adote novas estratégias para tratar novas situações e reagir às mudanças de contexto no ambiente da grade. O AGST implementa um mecanismo de transferência de estado durante a substituição de componentes em adaptações estruturais, e oferece um mecanismo que trata a sincronização entre a execução das ações de reconfiguração e a execução funcional dos elementos gerenciados.

5.2 Implementação do Modelo de Simulação da Abordagem Proposta

Nesta Seção veremos como foi implementado o mecanismo de gerenciamento de aplicações em *desktop grids* oportunistas proposto neste trabalho de mestrado. Esse mecanismo foi implementado e avaliado utilizando o simulador de grades AGST [18, 19]. Dentre os componentes que este mecanismo possui, o componente principal é a heurística de escalonamento desenvolvida para escalonar as aplicações com base nas restrições de tempo impostas pelo usuário. Esse algoritmo considera que existem duas classes de aplicações: *soft-deadline*, que designa as aplicações que possuem restrições quanto ao tempo máximo em que devem ser executadas; *nice*, que designa aplicações que não possuem restrições de tempo de execução.

Para a implementação do modelo de simulação do mecanismo proposto neste trabalho foram utilizados componentes do ciclo de gerenciamento autônomo

MAPE-K definido e implementado no simulador AGST. Foram estendidos os componentes de monitoramento e sensoriamento do ciclo MAPE-K para a implementação das facilidades de monitoramento do progresso de execução das tarefas e do tempo médio entre falhas apresentados pelos recursos da grade. Foi reutilizada a abordagem autonômica de tolerância a falhas na execução de aplicações da grade baseada no uso de *checkpoints* e implementada por Viana et al. [52, 53] no simulador AGST. Por fim, foi implementado a heurística de escalonamento proposta e apresentada na Seção 4.1.4. A Figura 5.2 mostra os componentes utilizados neste modelo de simulação, cuja implementação será detalhada a seguir.

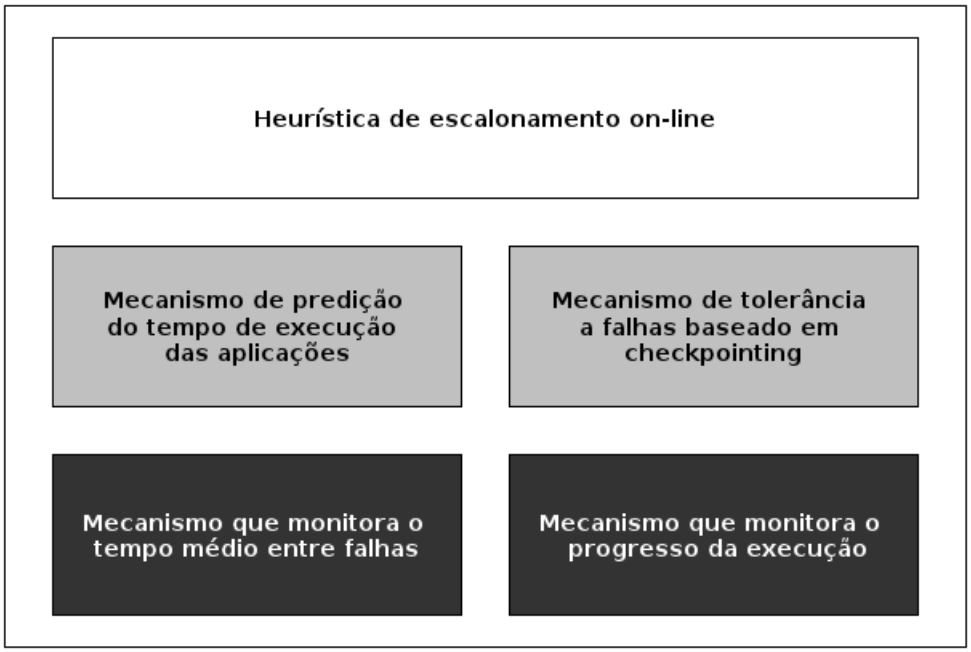

Figura 5.2: Componentes do modelo de simulação.

5.2.1 Mecanismos de Monitoramento

Para obter diversas informações sobre o contexto do ambiente de execução da grade foram desenvolvidos sensores e monitores, utilizando como base as classes fornecidas pelo AGST. Os sensores foram desenvolvidos a partir de instâncias da classe `Sensor`. Os monitores recebem uma referência ao sensor ao qual será periodicamente solicitado o valor de uma propriedade correspondente a algum tipo de informação de contexto do ambiente da grade. Foram implementados dois componentes de monitoramento: (a) `TaskMonitor`: componente que monitora o progresso da execução das tarefas; e (b) `MTBFMonitor`: componente que monitora o tempo médio

entre falhas de cada recurso. O diagrama de classes do monitores podem ser vistas na Figura 5.3.

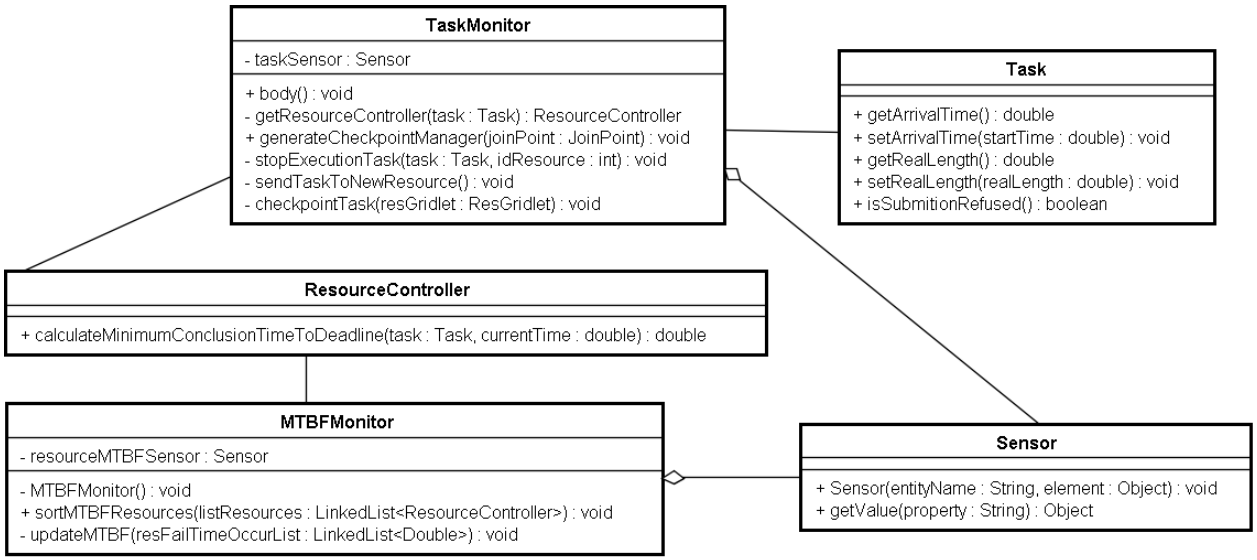

Figura 5.3: Diagrama de Classes dos Monitores.

Progresso da Execução de Tarefas

Para realizar o monitoramento do progresso da execução das tarefas, os sensores coletam dados das aplicações que estão executando nos recursos, esses dados são: (a) tamanho da aplicação em milhares de instruções; e (b) capacidade de cada recurso expressa em milhares de instruções por segundo. O monitor do progresso de execução das tarefas, chamado `TaskMonitor`, tem por função obter e processar essas informações sobre o andamento das aplicações que estão executando nos recursos. Essas informações são obtidas através de monitoramento periódico, cujo intervalo pode ser configurado pelo usuário. A classe `TaskMonitor` permite que, após receber os dados de monitoramento coletados pelos sensores, possa ser realizado um processamento sobre esses dados. O processamento consiste em verificar o andamento das tarefas em execução no recurso, e, caso o recurso não possa mais cumprir o *deadline* da tarefa será realizado uma busca por outro recurso que possa cumprir seu prazo de execução. Para isso, foi implementado o método `body()`.

Tempo Médio entre Falhas de Recursos

O monitor do tempo médio entre falhas é utilizado para ordenar os recursos dinamicamente pelo MTBF. Para implementar a ordenação dos recursos são

necessárias informações sobre o tempo médio entre falhas de cada recurso. Para obter essas informações foi implementado um monitor chamado `MTBFMonitor`, cuja função é obter e processar informações sobre o histórico da ocorrência de falhas dos recursos. Essas informações são obtidas através do monitoramento periódico, cujo intervalo pode ser configurado pelo usuário. O sensor `resourceMTBFSensor` obtém do `ResourceDataStorage` uma lista contendo as ocorrências de falhas dos recursos. O `MTBFMonitor` obtém esses dados desse sensor e calcula o tempo médio entre falhas de cada recurso da grade no monitor. Esse cálculo é feito somando-se o MTBF ao tempo de ocorrência da última falha de cada recurso.

5.2.2 Mecanismo de Predição

O mecanismo de predição utilizado neste trabalho de dissertação foi implementado em trabalhos anteriores, como foi dito na Seção 4.1. Nesta Seção veremos sucintamente como foi desenvolvido o mecanismo de predição. O mecanismo de predição foi implementado utilizando diretamente os recursos do simulador AGST adicionando erros no processo de estimativa do tempo de execução das aplicações em nós da grade, para isso foi implementado os métodos das classes `ResourceController` e `MathOGST` vistos na Figura 5.4.

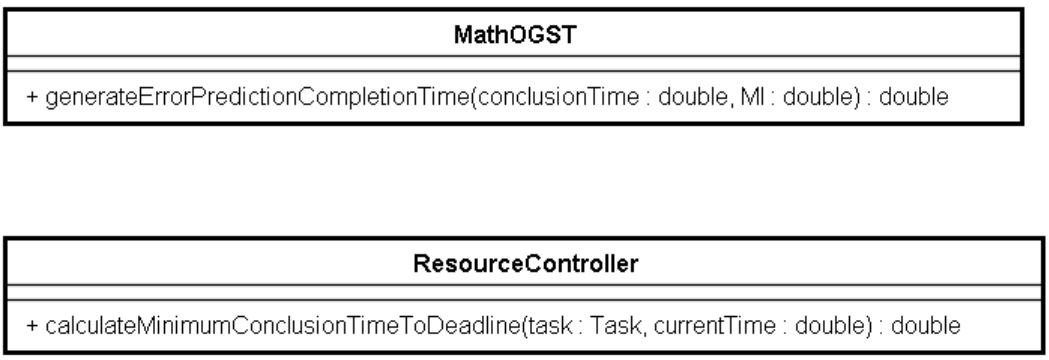

Figura 5.4: Classes que implementam o Mecanismo de Predição.

Para realizar a predição, o tamanho da aplicação é analisado e depois é estimado o tempo de execução de cada tarefa de acordo com a capacidade dos recursos da grade, recurso implementado pelo método `calculateMinimumConclusionTimeToDeadline` na classe `ResourceController`. Por fim, foi adotado uma taxa erro de 10% na predição tornando a simulação mais próximo de um ambiente real (não simulado), sendo

que a adição de erros no processo de estimativa foi implementado pelo método `generateErrorPredictionCompletionTime` na classe `MathOGST`.

5.2.3 Mecanismo de Tolerância a Falhas

O mecanismo de tolerância a falhas utilizado neste trabalho de dissertação foi implementado em trabalhos anteriores, como foi dito na Seção 4.1. Nesta Seção veremos sucintamente como foi desenvolvido o mecanismo de tolerância a falhas. O mecanismo adaptativo de tolerância a falhas foi implementado no simulador AGST com intuito de realizar *checkpointing* periódicos para as aplicações em nós da grade.

Para realizar ajustes no intervalo entre os *checkpoints* de cada tarefa é necessário coletar informações sobre o tempo médio entre falhas. Essa função é exercida pelo `resourceFailuresSensor`, um sensor acoplado também ao `ResourceDataStorage` que coleta desse componente uma lista contendo as ocorrências de falhas dos recursos. O `ResourceFailuresMonitor` obtém esses dados desse sensor e calcula o tempo médio entre falhas de cada recurso da grade através da função de pré-processamento. Em seguida, essas informações são enviadas ao analisador do ciclo local de *checkpointing*. O analisador recebe dados de contexto do monitor `ResourceFailuresMonitor` relativos ao percentual de falhas de recurso da grade. A partir dessas informações ele pode decidir se deve modificar ou não o intervalo de *checkpointing* das tarefas em execução na grade.

A fim de reduzir a sobrecarga desnecessária causada pelo *checkpointing*, o modelo proposto por Viana et al. [52, 53] e utilizado neste trabalho ajusta os intervalos entre os *checkpoints* de cada tarefa de acordo com a volatilidade do recurso em que está executando. Então, se os processos executam em recursos estáveis, esse ajuste é feito de forma que os intervalos entre *checkpoints* são alargados e, portanto, é reduzida a quantidade de vezes que são feitos. De outra forma, se os processos executam em recursos voláteis, os intervalos são reduzidos e, consequentemente, uma maior quantidade de *checkpoint* será feita, garantindo que sejam salvos estados mais próximos do estado de execução do momento da falha.

5.2.4 Heurística de Mapeamento Proposta

A heurística de escalonamento proposta neste trabalho, como já foi dito na Seção 4.1.4, é dinâmica e do tipo *on-line*. Assim, cada aplicação é mapeada imediatamente após sua submissão (caso haja recurso disponível), podendo mapear mais de uma tarefa para execução em um dado nó da grade. O mapeamento de tarefas da classe *soft-deadline*, é realizado levando-se em consideração recursos com maior intervalo médio entre falhas (MTBF) e cuja capacidade permita concluir a execução da tarefa dentro do prazo estipulado pelo usuário ao submeter a aplicação para execução na grade. Para que mudanças no contexto da execução das aplicações *soft-deadline* sejam percebidas, é necessário monitorar o progresso de execução de cada tarefa. Além disso, as informações sobre o MTBF de cada nó precisam ser monitoradas periodicamente para que a abordagem de escalonamento possa mapear as aplicações para aqueles considerados mais estáveis.

A heurística de escalonamento foi implementada no simulador AGST com o objetivo de atender o *deadline* das aplicações. Para implementar um algoritmo de escalonamento no simulador é preciso estender a classe abstrata `ScheduleStrategy`. Na classe abstrata `ScheduleStrategy` existe um método abstrato chamado `executeMapping` que recebe uma coleção de recursos (`ResourceController`) e retorna uma coleção de tarefas (`Task`) mapeadas para seus respectivos nós. A classe que estende `ScheduleStrategy` deve implementar este método que possui a seguinte assinatura: `executeMapping(LinkedList<ResourceController> resourceInfoList)` e que retorna uma coleção de tarefas (`LinkedList<Task>`) mapeadas para seus respectivos recursos. Para implementar este método utilizamos o método `canAchieveDeadline(resource, task, deadline)` que verifica se o prazo da tarefa pode ser atendida no recurso, caso o recurso possa cumprir o *deadline* da tarefa então é utilizado o método `maps(task, resource)` que realizará o mapeamento da tarefa para o recurso.

A combinação entre a abordagem de escalonamento e a abordagem de tolerância a falhas tem o objetivo de garantir a execução com sucesso das aplicações, mesmo na ocorrência de falhas de recursos. Porém, devido ao custo de tempo imposto pelo uso da técnica de *checkpointing* e a eventual variação na carga de trabalho local dos nós, considera-se como aptos para executar a tarefa os nós cujas capacidades de

processamento estimadas pelo mecanismo de predição adotado permitam concluir a execução da tarefa com certa margem de antecedência em relação ao *deadline* definido pelo usuário. Essa margem serve para garantir que a aplicação consiga executar no prazo determinado, ainda que eventuais atrasos ocorram. Caso a carga local de trabalho exceda a estimativa inicial, o mecanismo de acompanhamento do progresso da execução das tarefas é responsável por identificar a incapacidade do nó em cumprir o prazo estipulado e ressubmeter a tarefa para um novo escalonamento.

Na Figura 5.5 é apresentado o diagrama de classe da heurística de escalonamento proposta.

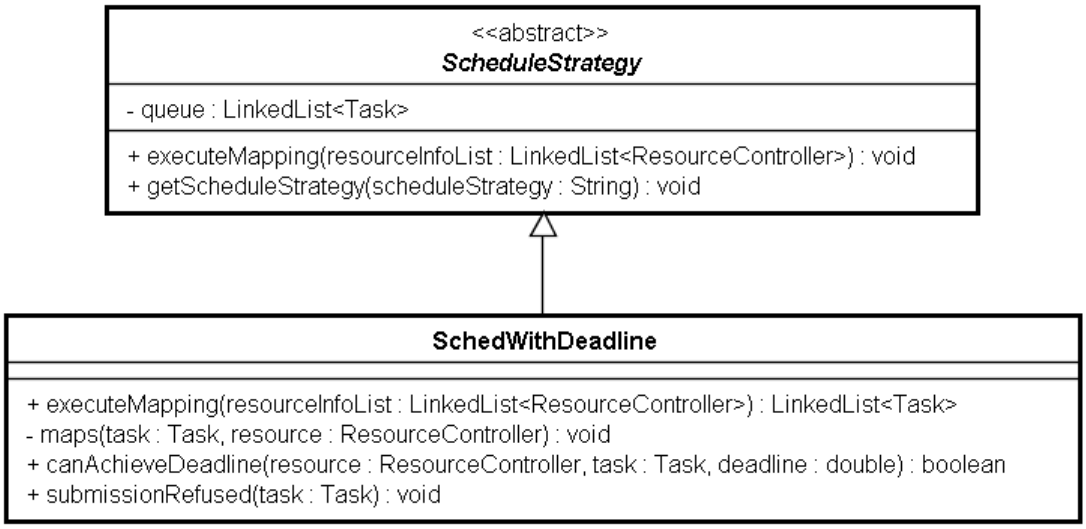

Figura 5.5: Diagrama de Classe da Heurística de Escalonamento.

5.3 Experimentos Realizados

A abordagem proposta foi avaliada através de simulações que levaram em consideração vários cenários típicos de grades oportunistas, utilizando-se como métrica a avaliação da quantidade de aplicações *soft-deadline* executadas dentro das restrições de tempo de execução informadas pelo usuário. Uma vez que este trabalho foi realizado no contexto do projeto InteGrade, foram comparados os resultados obtidos da nossa abordagem com o algoritmo de escalonamento InteGrade [28], que segue uma abordagem on-line.

O algoritmo de escalonamento InteGrade usa um filtro para selecionar os recursos com base nas restrições e preferências fornecidas pelos usuários durante o processo de submissão das aplicações. As restrições definem requisitos mínimos para a seleção das máquinas, tais como plataformas de hardware e software e requisitos de recursos, tais como requisitos mínimos de memória. As preferências definem a ordem usada para escolher os recursos, como por exemplo, executar as aplicações em uma CPU mais rápida em vez de uma mais lenta. As tarefas que compõem a aplicação são então mapeadas para os nós de acordo com a lista ordenada de recursos. Se os requisitos e preferências não são especificados, o algoritmo mapeia as tarefas para recursos da grade escolhidos aleatoriamente. O algoritmo pode mapear mais de uma tarefa por nó.

Para a realização das simulações, foi utilizado o AGST, um simulador de eventos discretos orientado a objetos. O simulador fornece, entre outras, ferramentas para modelagem de recursos da grade e suas interconexões de rede, aplicações da grade e suas submissões, a ocorrência de falhas de recursos, a carga de trabalho local dos recursos, o uso de *workload* e traces de falhas que seguem o padrão SWF e o FTA, e um modelo de banco de dados para armazenar dados gerados de simulações relevantes. No entanto, a contribuição mais importante do AGST é a definição e implementação de um modelo de simulação com base no ciclo de gerenciamento autônomo MAPE-K, que pode ser usado para simular o monitoramento, análise e planejamento, controle e execução de funções, permitindo a simulação de uma grade de computadores autonômica. Esta é uma característica importante, já que adotou-se neste trabalho um mecanismo de tolerância a falhas autonômico.

5.3.1 Avaliação sem Falhas de Recursos

Foi simulado um ambiente de grade oportunista composto por 100 máquinas em um mesmo domínio administrativo, interconectadas de forma homogênea por uma rede de 100 *Megabit per second* (Mbps). Nesse ambiente, o poder de processamento médio definido foi equivalente a um Pentium IV com 2,4 *Gigahertz* (GHz) (2.770 MIPS, tendo-se por base o benchmark TSCP[5]), considerado um valor representativo para computadores pessoais. A fim de levar em consideração a heterogeneidade dos recursos no ambiente, os nós da grade foram gerados sinteticamente através de uma distribuição uniforme. Foram realizadas várias simulações considerando dois fatores heterogeneidade: $U(1.385; 4.155)$ MIPS e $U(791; 4.746)$ MIPS. Utilizando o primeiro fator de heterogeneidade, o poder de processamento da máquina mais rápida é 3 vezes maior que o poder de processamento da maquina mais lenta. Já no segundo fator, a diferença é de 6 vezes.

As simulações levaram em consideração a existência de carga de trabalho local (*hostload*) nos recursos da grade. O modelo de carga de trabalho (*workload*) definida baseou-se em Conde [12]. Nesse trabalho foram coletados registros do uso de recursos (CPU e memória) de diversas máquinas pertencentes aos laboratórios do Departamento de Ciência da Computação da Universidade de São Paulo, armazenados em arquivos de *trace*. Através da leitura e análise desses arquivos, foi possível simular as cargas de trabalho tanto para dias de semana, quanto para finais de semana. Foi desenvolvido um aplicativo que gera vetores de carga de trabalho, cada um com 24 posições representando as 24 horas do dia. Os vetores foram passados por parâmetros para o AGST, que simula a carga de trabalho dos nós.

Em um primeiro conjunto de simulações livre de falhas, foram levados em consideração a execução de 770 aplicações regulares para cada experimento, geradas sinteticamente variando seu tamanho (em termos de milhões de instruções) através de uma distribuição uniforme $U(39.888 \times 10^3; 159.552 \times 10^3)$ MI. Considerando o poder de processamento médio utilizado para as máquinas da grade da grade (2.770 MIPS), cada aplicação levaria, aproximadamente, de 4 a 16 horas para ser executada. Foram simuladas as seguintes taxas de chegada das aplicações por segundos: 0,002 (0,12 aplicações por minuto); 0,004 (0,24 aplicações por minuto) e 0,006 (0,36 aplicações por

[5] `http://home.comcast.net/~tckerrigan/bench.html`

minuto). Durante as simulações, foram variadas também a quantidade de aplicações *soft-deadline* que compõem o conjunto de aplicações simuladas, utilizando os seguintes parâmetros: 25%, 50%, 75% e 100%. Para a modelagem e simulação dos eventos de chegada das aplicações em todas os experimentos foi utilizada a distribuição de *Poisson*, pois segundo Chwif e Medina [10] essa distribuição é uma das mais adequadas para essa finalidade. A Tabela 5.1 resume os parâmetros utilizados nas simulações realizadas.

Tabela 5.1: Resumo dos parâmetros de simulação

quantidade de máquinas	100 (nós)
fator de heterogeneidade	fator 3 = $U(1.385; 4.155)$ e fator 6 = $U(791; 4.746)$
aplicações regulares	770 (tarefas)
taxa de chegada das aplicações	0.12; 0.24 e 0.36 (app/min)
porcentagem de aplicações *soft*	25; 50; 75 e 100 (%)
tamanho das aplicações em MI	4 a 16 horas = $U(39.888 \times 10^3; 159.552 \times 10^3)$

Foram realizadas um total de 48 simulações diferentes, utilizando as duas estratégias de escalonamento (a nossa abordagem e o InteGrade), as três taxas de chegada das aplicações, os quatro valores da quantidade de aplicações *soft-deadline* e os dois fatores de heterogeneidade dos recursos. Para cada simulação, foram geradas 20 conjuntos de 770 aplicações, levando a um total de 960 experimentos.

A Figura 5.6 apresenta os resultados obtidos com as duas estratégias de escalonamento quando é utilizado uma taxa de chegada de 0,002 (0,12 aplicações por minuto) e um fator de heterogeneidade dos recursos igual a 3, onde o poder de processamento da máquina mais rápida é 3 vezes maior do que o da máquina mais lenta. Como se pode observar, a abordagem proposta neste trabalho atende a restrição de tempo de quase 100% das aplicações *soft-deadline* submetidas para a grade, mesmo quando 100% das aplicações submetidas são dessa classe. Isto é aproximadamente 25% melhor do que o algoritmo InteGrade, que realizou quase 80% dos prazos das aplicações. Neste caso, ambas as abordagens apresentaram um bom desempenho, já que a carga de trabalho da grade é relativamente baixa e a taxa de chegada não é tão alta a ponto de deixar os nós sobrecarregados.

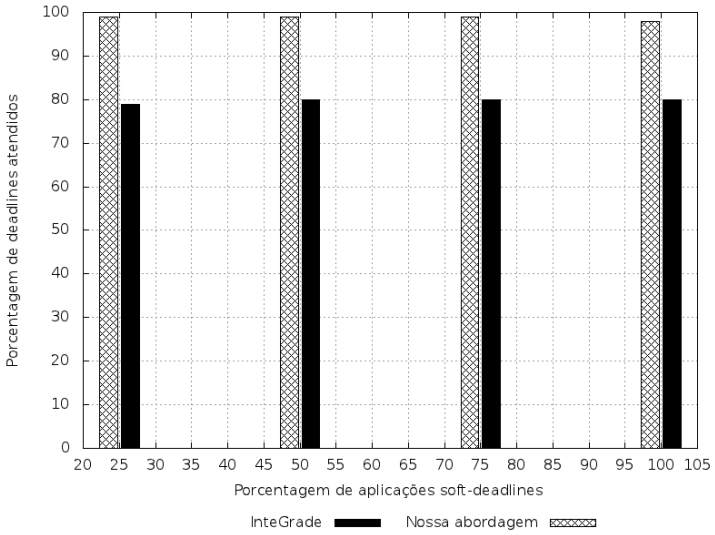

Figura 5.6: 1 aplicação a cada 0,002 segundos (0,12 min), fator de heterogeneidade igual a 3.

A Figura 5.7 mostra o resultado em um cenário onde a carga de trabalho da grade é maior, utilizando uma taxa de chegada de 0,006 (0,36 aplicações por minuto), mantendo o fator de heterogeneidade dos recursos, onde o poder de processamento da máquina mais rápida é 3 vezes maior do que o da máquina mais lenta. Como pode ser visto, neste caso o algoritmo InteGrade apresenta um pior resultado do que foi visto na simulação anterior, cumprindo apenas a 20% de prazos, quando 50% das aplicações submetidas foram *soft-deadline*. Nossa abordagem apresentou um resultado muito melhor, cumprindo quase 50% dos prazos submetidos, o que representa um ganho de 150%.

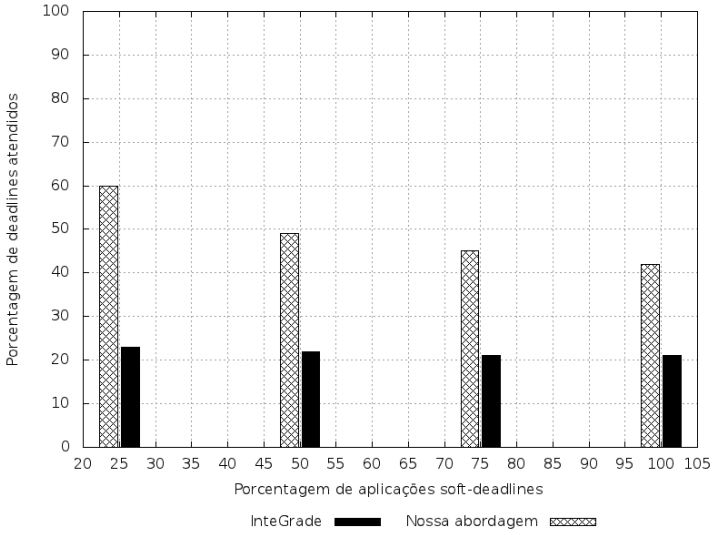

Figura 5.7: 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 3.

Em outro cenário simulado (Figura 5.8), manteve-se a taxa de chegada das aplicação de 0,006 (0,36 aplicações por minuto), alterando o fator de heterogeneidade de recursos, onde o poder de processamento da máquina mais rápida é 6 vezes maior do que o da máquina mais lenta. Neste caso, para um total de 50% de aplicações *soft-deadline* em cada conjunto de aplicações, a nossa abordagem conseguiu cumprir 60% dos prazos solicitados, enquanto o InteGrade cumpriu apenas 25%. Isto indica que nossa abordagem pode tirar proveito da maior heterogeneidade de recursos do ambiente para alcançar um desempenho superior.

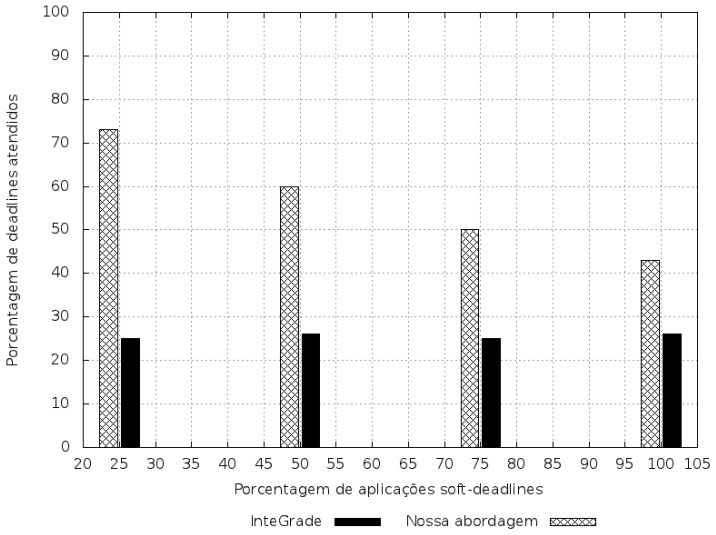

Figura 5.8: 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 6.

Analisando os resultados obtidos com as diversas simulações realizadas, pode-se concluir que o algoritmo proposto pelo fato de levar em consideração o *deadline* das aplicações consegue cumprir o prazo das aplicações muito melhor do que o InteGrade, considerando o objetivo de executar aplicações com restrições de tempo de execução em ambientes de grade oportunista.

5.3.2 Avaliação com Falhas de Recursos

Em um outro conjunto de simulações foram avaliados o desempenho das mesmas estratégias de escalonamento considerando os mesmos parâmetros utilizados nas simulações descritas na Seção anterior, sendo assim as taxas de chegada das aplicações por segundos são: 0,002 (0,12 aplicações por minuto); 0,004 (0,24 aplicações por minuto) e 0,006 (0,36 aplicações por minuto) e foram inseridas falhas de recursos, onde o valor do tempo médio entre falhas de recursos utilizados (MTBF) foi de

30 minutos. As máquinas da grade continua composta por 100 nós gerados da mesma forma descrita na Seção anterior. Nas simulações realizadas, o mecanismo de recuperação de falhas na execução de aplicações foi o *checkpoint*.

O tempo de indisponibilidade dos nós que falharam (*downtime*) seguiu uma distribuição exponencial, cuja média foi obtida a partir de uma distribuição uniforme $U(0,166; 30,00)$ horas. A cada evento de recuperação de nó foi gerado uma nova média. O objetivo dessa abordagem foi simular tanto falhas de curta quanto de longa duração, gerando-se uma quantidade maior de falhas de curta duração. Em um ambiente de grade oportunista, falhas de curta duração, tais como o reinício de máquinas por seu usuário local ou causadas por oscilação/falta de corrente elétrica são bem mais frequentes que as de longa duração, como as causadas por falhas no *hardware* das máquinas, por exemplo.

Um primeiro conjunto de simulações utilizou o mesmo ambiente de grade composto por 100 nós gerado para as simulações anteriores. Novamente foram geradas sinteticamente 770 aplicações regulares, com uma variação de tamanho (em termos de milhões de instruções) de $U(39.888 \times 10^3; 159.552 \times 10^3)$ MI, que levariam de 4 a 16 horas para serem executadas, considerando o poder de processamento médio utilizado para os nós da grade (2.770 MIPS).

Foram realizadas simulações considerando as três taxas de chegada de chegada de aplicações por minuto: 0,002 (0,12 aplicações por minuto); 0,004 (0,24 aplicações por minuto) e 0,006 (0,36 aplicações por minuto). Para cada taxa de chegada, foi inserido falhas de recursos com o valor do MTBF de 30 minutos. Dessa forma, considerando as 2 estratégias de escalonamento, as 3 taxas de chegada, os 4 valores da quantidade de aplicações *soft-deadline* e os 2 fatores de heterogeneidade dos recursos, foram realizadas 48 simulações, executadas 20 vezes, gerando um total de 960 experimentos.

A Figura 5.9 apresenta os resultados obtidos com as duas estratégias de escalonamento quando é utilizado uma taxa de chegada de 0,002 (0,12 aplicações por minuto) e um fator de heterogeneidade dos recursos igual a 3, onde o poder de processamento da máquina mais rápida é 3 vezes maior do que o da máquina mais lenta. As aplicações tiveram suas execuções reescalonadas em um outro nó da grade quando o nó em que executavam falhava. Como se pode ver, a abordagem proposta

neste trabalho atende a restrição de tempo melhor que o algoritmo InteGrade mesmo com as falhas dos recursos. Neste caso, a carga de trabalho da grade é relativamente baixa e a taxa de chegada não é tão alta a ponto de deixar os nós sobrecarregados.

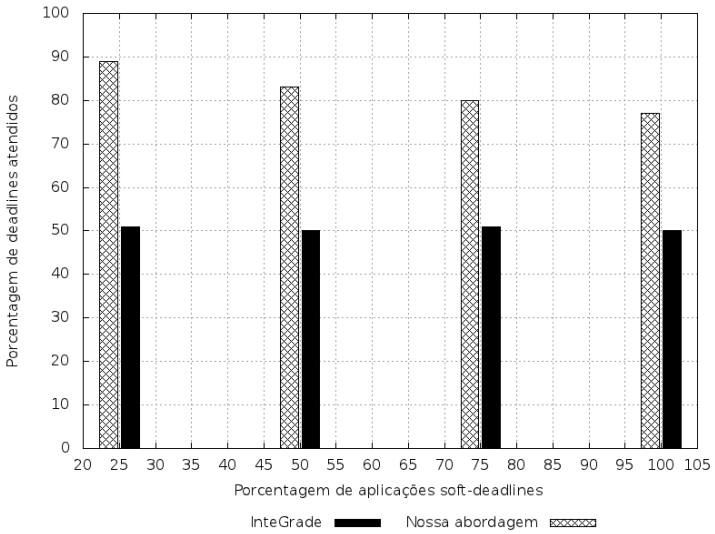

Figura 5.9: 1 aplicação a cada 0,002 segundos (0,12 min), fator de heterogeneidade igual a 3.

A Figura 5.10 mostra o resultado em um cenário com falhas onde a carga de trabalho da grade é maior, utilizando uma taxa de chegada de 0,006 (0,36 aplicações por minuto), mantendo o fator de heterogeneidade dos recursos, onde o poder de processamento da máquina mais rápida é 3 vezes maior do que o da máquina mais lenta. Como pode ser visto, neste caso o algoritmo InteGrade apresenta um pior resultado do que foi visto na simulação anterior.

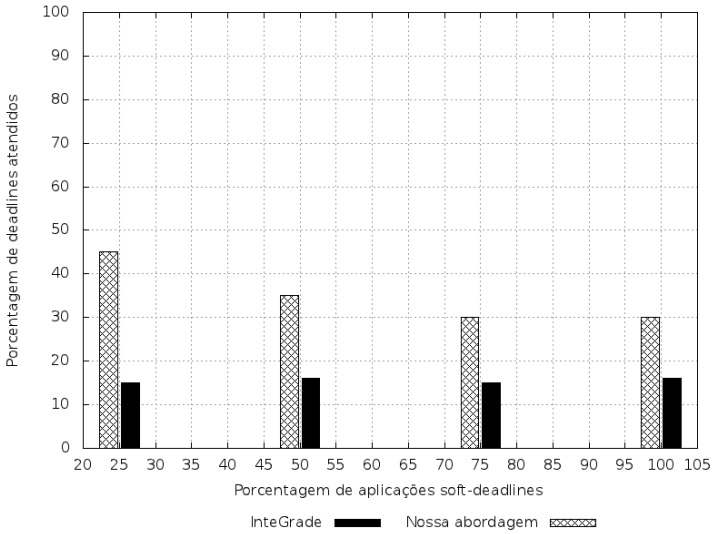

Figura 5.10: 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 3.

Em um outro conjunto de simulações com falhas (Figura 5.11), foi avaliado o desempenho das mesmas estratégias de escalonamento mantendo a taxa de chegada das aplicação de 0,006 (0,36 aplicações por minuto), alterando o fator de heterogeneidade de recursos, onde o poder de processamento da máquina mais rápida é 6 vezes maior do que o da máquina mais lenta. Neste caso, para um total de 50% de aplicações *soft-deadline* em cada conjunto de aplicações, a nossa abordagem conseguiu cumprir mais de 40% dos prazos solicitados, enquanto o InteGrade cumpriu apenas 17%. Isto indica que nossa abordagem pode tirar proveito da maior heterogeneidade de recursos do ambiente para alcançar um desempenho superior.

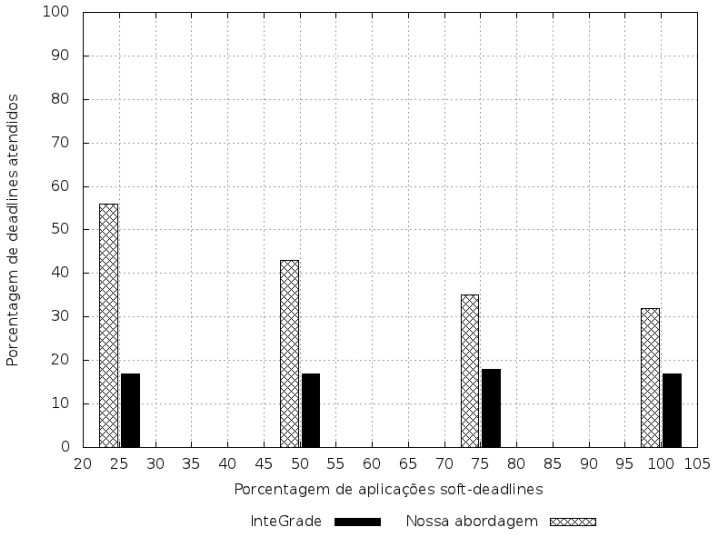

Figura 5.11: 1 aplicação a cada 0,006 segundos (0,36 min), fator de heterogeneidade igual a 6.

Analisando os resultados obtidos com as diversas simulações com falhas realizadas, pode-se concluir que diversos fatores relacionados ao ambiente de execução, como a taxa de chegada das aplicações, o tamanho das tarefas, o tempo médio entre falhas de nós, e o tempo de recuperação dessas falhas interferem no desempenho das estratégias de escalonamento. Assim o algoritmo proposto pelo fato de levar em consideração o *deadline* das aplicações consegue cumprir o prazo das aplicações muito melhor do que o InteGrade, considerando o objetivo de executar aplicações com restrições de tempo de execução em ambientes de grade oportunista.

5.4 Conclusões

Este capítulo apresentou a avaliação da abordagem proposta nesta dissertação de mestrado. Foi Visto que a avaliação foi realizada utilizando-se uma ferramenta de simulação desenvolvida para ambientes de grades de computadores, o *Autonomic Grid Simulation Tool* (AGST).

Foi apresentado também a implementação do modelo de simulação do mecanismo de gerenciamento de aplicações em *desktop grids* oportunistas. Foi mostrada a implementação da heurística de escalonamento que compõe o mecanismo de gerenciamento de aplicações proposto. Finalmente, foi apresentado a avaliação do algoritmo proposto neste trabalho utilizando um ambiente de grade simulado que foi provido pelo simulador AGST para ambientes de grades oportunistas levando em consideração também a carga de trabalho local proveniente de aplicações executadas pelos usuários locais das máquinas.

A análise dos resultados das simulações descritas neste capítulo enfatizaram que as heurísticas de escalonamento executam diferentemente sobre diversas condições do ambiente da grade. No entanto, o algoritmo proposto se destaca em relação ao algoritmo InteGrade por levar em consideração o *deadline* das aplicações conseguindo cumprir o prazo das aplicações muito melhor do que o InteGrade tanto em um ambiente livre de falhas quanto em um ambiente com falhas, considerando o objetivo de executar aplicações com restrições de tempo de execução em ambientes de grade oportunista.

A métrica utilizada como critério de avaliação foi a porcentagem de aplicações executadas com sucesso dentro do prazo estipulado pelo usuário. Os resultados das simulações permitiram concluir que o mecanismo de escalonamento e a abordagem para tolerância a falhas baseada em *checkpointing*, quando adequadamente combinados, compõem um modelo de gerenciamento que permite executar eficientemente aplicações com prazo de conclusão, mesmo em um ambiente tão dinâmico quanto o das *desktop grids* oportunistas. O mecanismo para o gerenciamento da execução de aplicações foi avaliado em diversos cenários, levando-se em consideração características típicas de *desktop grids* oportunistas, apresentando resultados significativamente melhores do que os apresentados pelo mecanismo

tradicional de escalonamento de aplicações utilizado pelo *middlware* InteGrade, o qual serviu como base para a comparação dos resultados.

6 Conclusão e Trabalhos Futuros

Em uma grade oportunista, ciclos ociosos de um grande número de computadores pessoais integrados são utilizados para a execução de aplicações de computação intensiva. O ambiente de execução de grades de computadores oportunistas é tipicamente dinâmico, heterogêneo e imprevisível, além de sujeito a falhas frequentes. Os recursos de uma grade oportunista podem também ser utilizados a qualquer momento para a execução de tarefas locais, tornando difícil a previsão do término das tarefas em execução nos nós da grade. Estas características dificultam a execução de aplicações para as quais existem restrições de tempo para sua conclusão.

Este trabalho apresentou uma nova abordagem para o gerenciamento de aplicações que possuam restrições de tempo de execução brandas (*soft-deadlines*) desenvolvido especificamente para ambientes de grades oportunistas. A abordagem proposta é composta por um mecanismo de predição do tempo de execução das aplicações nos nós da grade; uma heurística de escalonamento do tipo *on-line*; um mecanismo que acompanha o progresso da execução das tarefas em cada nó da grade e um mecanismo adaptativo de tolerância a falhas baseado em *checkpoint*. Quando adequadamente combinados, estes mecanismos compõem um modelo de gerenciamento que permite executar eficientemente aplicações com prazo de conclusão, mesmo em um ambiente tão dinâmico quanto o de grades oportunistas. A abordagem proposta foi devidamente avaliada em um ambiente simulado, apresentando resultados experimentais significativamente melhores do que o uso de mecanismos tradicionais de escalonamento de aplicações para grades de computadores.

As principais contribuições deste trabalho são:

- A investigação do estado da arte em escalonamento e gerenciamento de aplicações para grades de computadores;
- A concepção de um mecanismo para o gerenciamento da execução de aplicações que possuam restrições de tempo de execução brandas;
- O desenvolvimento de um modelo de simulação do mecanismo proposto;

- A avaliação do modelo proposto, levando-se em consideração diversos cenários típicos de grades oportunistas, a partir do modelo de simulação desenvolvido.

6.1 Trabalhos Futuros

A partir deste trabalho inicial, foram identificados algumas possibilidades de trabalhos futuros que poderiam ser desenvolvidos, tais como:

- Realizar modificações no mecanismo proposto para que ele leve em consideração outras classes de aplicações típicas de grades de computadores, tais como: *Bag-of-Tasks*, *Workflow* e *Parameter Sweep*;
- Integrar ao middleware InteGrade o mecanismo de gerenciamento da execução de aplicações proposto neste trabalho;
- Avaliar o impacto do uso de mecanismos de predição da disponibilidade futura de recursos da grade, como o disponibilizado pelos componentes *Local Usage Pattern Analyser* (LUPA) e *Global Usage Pattern Analyser* (GUPA) do middleware InteGrade, ao processo de escalonamento e re-escalonamento de aplicações adotado no mecanismo proposto neste trabalho.

Referências Bibliográficas

[1] A. H. M. Aliaga. Analise de desempenho de algoritmos de escalonamento em simuladores de grades. 2009.

[2] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. 2003.

[3] M. Baker, R. Buyya, and D. Laforenza. The grid: International efforts in global computing. *In Proc. of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2000.

[4] F. Berman, G. Fox, and A. Hey. *Grid computing: making the global infrastructure a reality*. Wiley series on parallel and distributed computing. Wiley, 2003.

[5] R. Buyya. Understanding the grid. *Grid Computing Planet Conference*, 2002.

[6] R. Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Software: Practice and Experience*, 35(5):491–512, April 2005.

[7] R. Buyya, S. Venugopal, R. Ranjan, and C. S. Yeo. The gridbus middleware for market-oriented computing. In R. Buyya and K. Bubendorfer, editors, *Market-Oriented Grid and Utility Computing*, chapter 26, pages 589–622. John Wiley and Sons, Hoboken, NJ, USA, 2009.

[8] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing Workshop*, HCW '00, pages 349–364, Washington, DC, USA, 2000. IEEE Computer Society.

[9] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2):141–154, February 1988.

[10] A. C. Chwif, Leonardo. e Medina. *Modelagem e Simulação de Eventos Discretos. Teoria & Aplicações. 2ª Ed*. Edição do Autor, São Paulo, Brazil, 2 edition, 2007. ISBN 8590597822.

[11] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[12] D. Conde. Análise de padrões de uso em grades computacionais. Master's thesis, Departamento de Ciência da Computação - Universidade de São Paulo, Brasil, SP, Janeiro 2008. Acessado em: 28/11/2011.

[13] G. Cunha Filho. OGST (Opportunistic Grid Simulation Tool): Uma Ferramenta de Simulação para Avaliação de Estratégias de Escalonamento de Aplicações em Grades Oportunistas. Master's thesis, Universidade Federal do Maranhão, São Luís, MA, Brasil, 2009.

[14] N. A. da Nóbrega Júnior. Avaliação de heurísticas de escalonamento de aplicações bag-of-tasks em grids computacionais adaptativas à disponibilidade de informação. Master's thesis, Universidade Federal de Campina Grande, Campina Grande, Paraíba, Brasil, Agosto 2006.

[15] D. da Silva, W. Cirne, and F. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 169–180. Springer Berlin / Heidelberg, 2003.

[16] F. J. da Silva e Silva, F. Kon, A. Goldman, M. Finger, R. Y. de Camargo, F. C. Filho, and F. M. Costa. Application execution management on the integrade opportunistic grid middleware. *Journal of Parallel and Distributed Computing*, 70(5):573 – 583, May 2010.

[17] L. de Assis. Uma heurística de escalonamento adaptativa à disponibilidade da informação para aplicações bag-of-tasks data-intensive em grids computacionais. Master's thesis, Universidade Federal de Campina Grande, Campina Grande, Paraíba, Brasil, Setembro 2009.

[18] B. de Tácio Pereira Gomes. Agst (autonomic grid simulation tool): uma ferramenta para modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores. Dissertação, Universidade Federal do Maranhão, São Luís, Maranhão, 2012.

[19] B. de Tácio Pereira Gomes and F. J. da Silva e Silva. Agst - autonomic grid simulation tool - a simulator of autonomic functions based on the mape-k model. In *SIMULTECH*, pages 354–359. SciTePress, 2011.

[20] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical Report 2006-504, School of Computing, Queen's University, Kingston, Ontario, Janeiro 2006.

[21] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[22] G. C. Filho and F. J. da Silva e Silva. Ogst: An opportunistic grid simulation tool. In *LAGrid 2008: 2nd International Workshop Latin American Grid*, Mato Grosso do Sul, Campo Grande, Agosto 2008.

[23] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the i-way high-performance distributed computing experiment. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 562–, Washington, DC, USA, 1996. IEEE Computer Society.

[24] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[25] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*, chapter Globus: A Toolkit-Based Grid Architecture, pages 259–278. Morgan Kaufmann Publisher, 1999.

[26] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputing Applications*, 15(3):200–222, 2001.

[27] A. Goldchleger. Integrade: Um sistema de middleware para computação em grade oportunista. Master's thesis, Universidade de São Paulo, 2004.

[28] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[29] B. d. P. Gomes, G. Cunha Filho, I. Ramos Campos, J. Figueredo Goncalves, and F. Jose da Silva e Silva. Scheduling strategies evaluation for opportunistic grids. In *Proceedings of the 2010 11th Symposium on Computing Systems*, WSCAD-SCC '10, pages 88–95, Washington, DC, USA, 2010. IEEE Computer Society.

[30] L. Gong, X.-H. Sun, and E. F. Watson. Performance modeling and prediction of nondedicated network computing. *IEEE Transactions on Computers*, 51(9):1041–1055, September 2002.

[31] A. S. Grimshaw, W. A. Wulf, and C. The Legion Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40:39–45, January 1997.

[32] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM (JACM)*, 24(2):280–289, April 1977.

[33] IBM. *An architectural blueprint for autonomic computing*, 2003.

[34] R. Jain. *The art of computer systems performance analysis*. Wiley, 1991.

[35] L. J. d. O. D. R. B. José Nelson Falavinha Junior, Aleardo Manacero Júnior. Avaliação de algoritmos de escalonamento em grids para diferentes configurações de ambiente. *Anais do Wperformance 2007 - XXVII Congresso da SBC*, 2007.

[36] K. H. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 541–548, Washington, DC, USA, May 2007. IEEE Computer Society.

[37] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'10)*, pages 398–407, Melbourne, Australia, Maio 2010. IEEE Computer Society.

[38] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computin. In *Software - Practice and Experience*, volume 32, pages 135–164. February 2002.

[39] Y. Liu. Survey on grid scheduling. for phd qualifying exam, Department of Computer Science, University of Iowa, April 2004.

[40] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, Novembro 2003.

[41] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pages 30–, Washington, DC, USA, 1999. IEEE Computer Society.

[42] R. Murch. *Autonomic Computing*. IBM Press / Prentice-Hall, 1nd edition, 2004.

[43] M. W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, Performance '87, pages 529–544, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[44] M. W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Perform. Eval.*, 12(4):269–284, August 1991.

[45] Object Management Group. *Trading Object Service Specification, Junho de 2000*, 1.0 edition, June 2000. OMG document formal/00-06-27.

[46] P. M. P. Domingues and L. Silva. Resource usage of windows computer laboratories. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, ICPPW '05, pages 469–476, Washington, DC, USA, 2005. IEEE Computer Society.

[47] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies*

for Parallel Processing, volume 3277 of *Lecture Notes in Computer Science*, pages 54–103. Springer Berlin / Heidelberg, 2005.

[48] S. S. Sathya and K. S. Babu. Survey of fault tolerant techniques for grid. *Computer Science Review*, 4(2):101–120, May 2010.

[49] J. M. Schopf. Ten actions when grid scheduling: the user as a grid scheduler. *Grid resource management: state of the art and future trends*, pages 15–23, 2004.

[50] X.-H. Sun and M. Wu. Grid harvest service: A system for long-term, application-level task scheduling. *Parallel and Distributed Processing Symposium, International*, 0:25a, april 2003.

[51] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[52] A. E. B. Viana. Uma abordagem autonômica para tolerância a falhas na execução de aplicações em desktop grids. Dissertação, Universidade Federal do Maranhão, São Luís, Maranhão, 2011.

[53] A. E. B. Viana, B. de Tácio P. Gomes, J. F. Gonçalves, L. R. Coutinho, and F. J. da Silva e Silva. Design and evaluation of autonomic fault tolerance strategies using the agst autonomic grid simulator. In *LatinAmerican Conference on High Performance and Distributed Computing (CLCAR '11)*, Colina, Mexico, Sep 2011.

[54] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Sci. Program.*, 14(3,4):217–230, December 2006.

[55] Y. Zhu. A survey on grid scheduling systems. Technical report, Computer Science Department, University of Science and Technology, Hong Kong, 2003.